Build Better Applications with Coding and Design Patterns

JavaScript Patterns

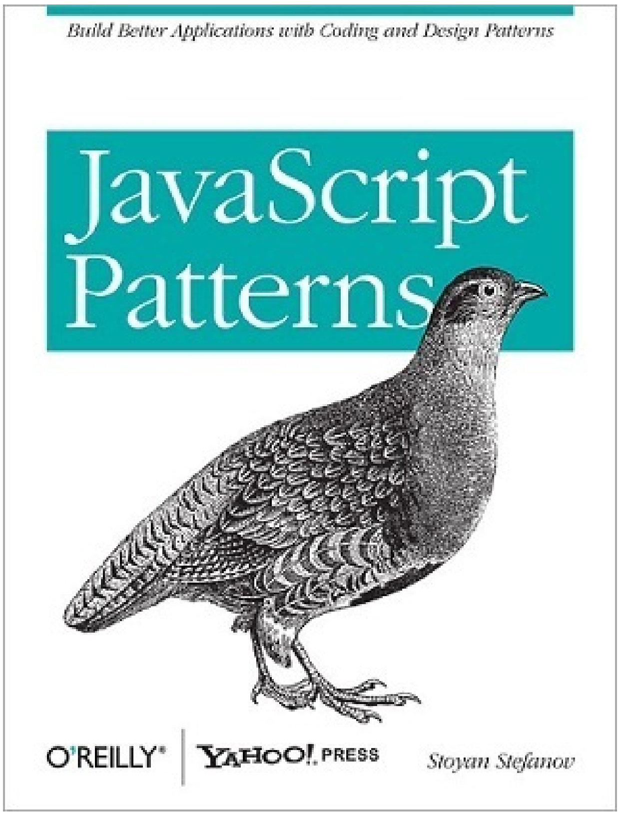


O'REILLY® | YAHOO! PRESS

Stoyan Stefanov

目錄

JavaScript Patterns



“JavaScript patterns”中译本 - 《JavaScript 模式》

- 作者：Stoyan Stefanov
- 翻译：拔赤、goddyzhao、TooBug

偷懒是程序员的优良品质，模式则是先人们总结的偷懒招式。Stoyan Stefanov 的这本书，从 JavaScript 的实际使用场景出发，提炼了不少可以让前端们偷懒的实用招式。模式的探索、创新，将永远是程序员自我提升的一条修炼之道。值得一读。

目录

第二章 高质量JavaScript基本要点

第三章 直接量和构造函数

第四章 函数

第五章 对象创建模式

第六章 代码复用模式

第七章 设计模式

第八章 DOM和浏览器模式

    - 预加载
- 小节

索引

第一章 概述

JavaScript是一门Web开发语言。起初只是用来操作网页中为数不多的元素（比如图片和表单域），但谁也没想到这门语言的成长是如此迅速。除了适用于客户端浏览器编程，如今JavaScript程序可以运行于越来越多的平台之上。你可以用它来进行服务器端开发（使用.Net或Node.js）、桌面应用程序开发（运行于桌面操作系统）、以及应用程序扩展（Firefox插件或者Photoshop扩展）、移动终端应用和纯命令行的批处理脚本。

JavaScript同样是一门不寻常的语言。它没有类，许多场景中它使用函数作为一等对象。起初，许多开发者认为这门语言存在很多缺陷，但最近几年情况发生了微妙的变化。有意思的是，有一些老牌语言比如Java和PHP也已经开始添加诸如闭包和匿名函数等新特性，而闭包和匿名函数则是JavaScript程序员最愿意津津乐道的话题。

JavaScript十分灵活，可以用你所熟悉的其他任何编程语言的编程风格来写JavaScript程序。但最好的方式还是拥抱它所带来的变化、学习它所特有的编程模式。

模式

对“模式”的广义解释是“反复发生的事件或对象的固定用法...可以用来作为重复使用的模板或模型”（http://en.wikipedia.org/wiki/Pattern）。

在软件开发领域，模式是指常见问题的通用解决方案。模式不是简单的代码复制和粘贴，而是一种最佳实践，一种高级抽象，是解决某一类问题的范本。

识别这些模式非常重要，因为：

- 这些模式提供了经过论证的最佳实践，它可以帮助我们更好的编码，避免重复制造车轮。
- 这些模式提供了高一层的抽象，某个时间段内大脑只能处理一定复杂度的逻辑，因此当你处理更繁琐棘手的问题时，它会帮你理清头绪，你才不会被低级的琐事阻碍大脑思考，因为所有的细枝末节都可以被归类和切分成不同的块（模式）。
- 这些模式为开发者和团队提供了沟通的渠道，团队开发者之间往往是异地协作，不会有经常面对面的沟通机会。简单的代码编写技巧和技术问题处理方式的约定（代码注释）使得开发者之间的交流更加通畅。例如，“函数立即执行”用大白话表述成“你写好一个函数后，在函数的结束花括号的后面添加一对括号，这样能在定义函数结束后马上执行这个函数”（我的天）。

本书将着重讨论下面这三种模式：

- 设计模式（Design patterns）
- 编码模式（Coding patterns）

- 反模式（Antipatterns）

设计模式最初的定义是来自于“GoF”（四人组，94年版“设计模式”的四个作者）的一本书，这本书在1994年出版，书名全称是“设计模式：可复用面向对象软件基础”。书中列举了一些重要的设计模式，比如单体、工厂、装饰者、观察者等等。但适用于JavaScript的设计模式并不多，尽管设计模式是脱离某种语言而存在的，但通常会以某种语言做范例来讲解设计模式，这些语言多是强类型语言，比如C++和Java。有时直接将其应用于弱类型的动态语言比如JavaScript又显得捉襟见肘。通常这些设计模式都是基于语言的强类型特性以及类的继承。而JavaScript则需要某种轻型的替代方案。本书在第七章将讨论基于 JavaScript实现的一些设计模式。

编码模式更有趣一些。它们是JavaScript特有的模式和最佳实践，它利用了这门语言独有的一些特性，比如对函数的灵活运用，JavaScript编码模式是本书所要讨论的重点内容。

本书中你会偶尔读到一点关于“反模式”的内容，顾名思义，反模式具有某些负作用甚至破坏性，书中会顺便一提。反模式并不是bug或代码错误，它只是一种处理问题的对策，只是这种对策带来的麻烦远超过他们解决的问题。在示例代码中我们会对反模式做明显的标注。

JavaScript：概念

在正式的讨论之前，应当先理清楚JavaScript中的一些重要的概念，这些概念在后续章节中会经常碰到，我们先来快速过一下。

面向对象

JavaScript 是一门面向对象的编程语言，对于那些仓促学习JavaScript并很快丢掉它的开发者来说，这的确有点让人感到意外。你所能接触到的任何JavaScript代码片段都可以作为对象。只有五类原始类型不是对象，它们是数字、字符串、布尔值、null和undefined，前三种类型都有与之对应的包装对象（下一章会讲到）。数字、字符串和布尔值可以轻易的转换为对象类型，可以通过手动转换，也可以利用JavaScript解析器进行自动转换。

函数也是对象，也可以拥有属性和方法。

在任何语言中，最简单的操作莫过于定义变量。那么，在JavaScript中定义变量的时候，其实也在和对象打交道。首先，变量自动变为一个被称作“活动对象”的内置对象的属性（如果是全局变量的话，就变为全局对象的属性）。第二，这个变量实际上也是“伪对象”，因为它有自己的属性（属性特性），用以表示变量是否可以被修改、删除或在for-in循环中枚举。这些特性并未在ECMAScript3中作规定，而ECMAScript5中提供了一组可以修改这些特性的方法。

那么，到底什么是对象？对象能作这么多事情，那它们一定非常特别。实际上，对象是及其简单的。对象只是很多属性的集合，一个名值对的列表（在其他语言中可能被称作关联数组），这些属性也可以是函数（函数对象），这种函数我们称为“方法”。

关于对象还需要了解，我们可以随时随地修改你创建的对象（当然，ECMAScript5中提供了可阻止这些修改的API）。得到一个对象后，你可以给他添加、删除或更新成员。如果你关心私有成员和访问控制，本书中我们也会讲到相关的编程模式。

最后一个需要注意的是，对象有两大类：

- 本地对象（Native）：由ECMAScript标准规范定义的对象
- 宿主对象（Host）：由宿主环境创建的对象（比如浏览器环境）

本地对象也可以被归类为内置对象（比如Array，Date）或自定义对象（var o = {}）。

宿主对象包含window和所有DOM对象。如果你想知道你是否在使用宿主对象，将你的代码迁移到一个非浏览器环境中运行一下，如果正常工作，那么你的代码只用到了本地对象。

无类

在本书中的许多场合都会反复碰到这个概念。JavaScript中没有类，对于其他语言的编程老手来说这个观念非常新颖，需要反复的琢磨和重新学习才能理解JavaScript只能处理对象的观念。

没有类，你的代码变得更小巧，因为你不必使用类去创建对象，看一下Java风格的对象创建：

```
// Java object creation
HelloOO hello_oo = new HelloOO();
```

为了创建一个简单的对象，同样一件事情却重复做了三遍，这让这段代码看起来很“重”。而大多数情况下，我们只想让我们的对象保持简单。

在JavaScript中，你需要一个对象，就随手创建一个空对象，然后开始给这个对象添加有趣的成员。你可以给它添加原始值、函数或其他对象作为这个对象属性。“空”对象并不是真正的空，对象中存在一些内置的属性，但并没有“自有属性”。在下一章里我们对此作详细讨论。

“GoF”的书中提到一条通用规则，“组合优于继承”，也就是说，如果你手头有创建这个对象所需的资源，更推荐直接将这些资源组装成你所需的对象，而不推荐先作分类再创建链式父子继承的方式来创建对象。在JavaScript中，这条规则非常容易遵守，因为JavaScript中没有类，而且对象组装无处不在。

原型

JavaScript中的确有继承，尽管这只是一种代码重用的方式（本书有专门的一章来讨论代码重用）。继承可以有多种方式，最常用的方式就是利用原型。原型（prototype）是一个普通的对象，你所创建的每一个函数会自动带有prototype属性，这个属性指向一个空对象，这个空

对象包含一个constructor属性，它指向你新建的函数而不是内置的Object()，除此之外它和通过对象直接量或Object()构造函数创建的对象没什么两样。你可以给它添加新的成员，这些成员可以被其他的对象继承，并当作其他对象的自有属性来使用。

我们会详细讨论JavaScript中的继承，现在只要记住：原型是一个对象（不是类或者其他什么特别的东西），每个函数都有一个prototype属性。

运行环境

JavaScript程序需要一个运行环境。一个天然的运行环境就是浏览器，但这绝不是唯一的运行环境。本书所讨论的编程模式更多的和JavaScript语言核心（ECMAScript）相关，因此这些编程模式是环境无关的。有两个例外：

- 第八章，这一章专门讲述浏览器相关的模式
- 其他一些展示模式的实际应用的例子

运行环境会提供自己的宿主对象，这些宿主对象并未在ECMAScript标准中定义，它们的行为也是不可预知的。

ECMAScript 5

JavaScript语言的核心部分（不包含DOM、BOM和外部宿主对象）是基于ECMAScript标准（简称为ES）来实现的。其中第三版是在1999年正式颁布的，目前大多数浏览器都实现了这个版本。第四版已经废弃了。第三版颁布后十年，2009年十二月，第五版才正式颁布。

第五版增加了新的内置对象、方法和属性，但最重要的增加内容是所谓的严格模式（strict mode），这个模式移除了某些语言特性，让程序变得简单且健壮。比如，with语句的使用已经争论了很多年，如今，在ECMAScript5严格模式中使用with则会报错，而在非严格模式中则是ok的。我们通过一个指令来激活严格模式，这个指令在旧版本的语言实现中被忽略。也就是说，严格模式是向下兼容的，因为在不支持严格模式的旧浏览器中也不会报错。

对于每一个作用域（包括函数作用域、全局作用域或在eval()参数字符串的开始部分），你可以使用这种代码来激活严格模式：

```
function my() {
    "use strict";
    // rest of the function...
}
```

这样就激活了严格模式，函数的执行则会被限制在语言的严格子集的范围内。对于旧浏览器来说，这句话只是一个没有赋值给任何变量的字符串，因此不会报错。

按照语言的发展计划，未来将会只保留“严格模式”。因此，现在的ES5只是一个过渡版本，它鼓励开发者使用严格模式，而非强制。

本书不会讨论ES5新增特性相关的模式，因为在本书截稿时并没有任何浏览器实现了ES5，但本书的示例代码通过一些技巧鼓励开发者向新标准转变：

- 确保所提供的示例代码在严格模式下不包错
- 避免使用并明确指出弃用的构造函数相关的属性和方法，比如arguments.callee
- 针对ES5中的内置模式比如Object.create()，在ES3中实现等价的模式

JSLint

JavaScript是一种解释型语言，它没有静态编译时的代码检查，所以很可能将带有简单类型错误的破碎的程序部署到线上，而且往往意识不到这些错误的存在。这时我们就需要JSLint的帮助。

JSLint（http://jslint.com ）是一个JavaScript代码质量检测工具，它的作者是 Douglas Crockford，JSLint会对代码作扫描，并针对潜在的问题报出警告。笔者强烈推荐你在执行代码前先通过JSlint作检查。作者给出了警告：这个工具可能“会让你不爽”，但仅仅是在开始使用它的时候不爽一下而已。你会很快从你的错误中吸取教训，并学习这些成为一名专业的JavaScript程序员应当必备的好习惯。让你的代码通过JSLint的检查，这会让你对自己的代码更加有自信，因为你不用再去担心代码中某个不起眼的地方丢失了逗号或者有某种难以察觉的语法错误。

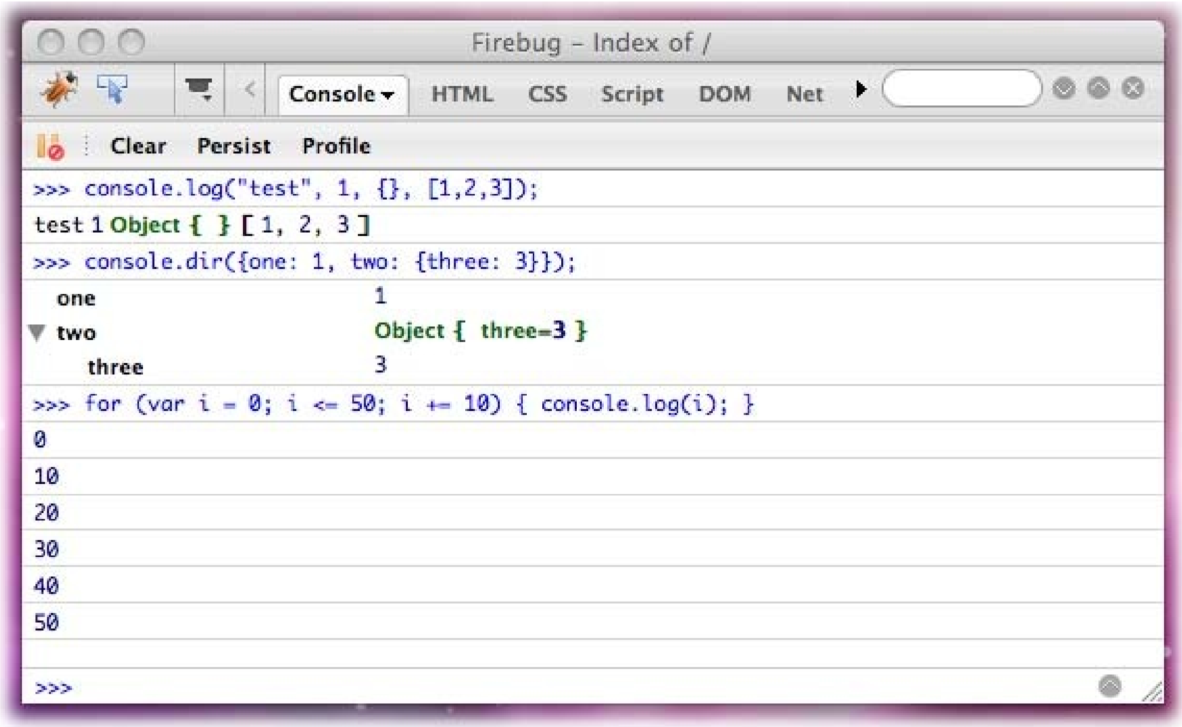
当开始下一章的学习时，你将发现JSLint会被多次提到。本书中除了讲解反模式的示例代码外（有清楚的注释说明）、所有示例代码均通过了JSLint的检查（使用JSLint的默认设置）。

控制台工具

console对象在本书中非常常见。这个对象并不是语言的一部分，而是运行环境的一部分，目前大多数浏览器也都实现了这个对象。比如在Firefox中，它是通过Firebug扩展引入进来的。Firebug控制台工具包含UI操作界面，可以让你快速输入并测试JavaScript代码片段，同样用它可以调试当前打开的页面（图1-1）。在这里强烈推荐使用它来辅助学习。在Webkit核心的浏览器（Safari和Chrome）也提供了类似的工具，可以监控页面情况，IE从版本8开始也提供了开发者工具。

本书中大多数代码都使用console对象来输出结果，而没有使用alert()或者刷新当前页面。因为用这种方法输出结果实在太简单了。

图 1-1 使用Firebug控制台工具



我们经常使用log()方法，它将传入的参数在控制台输出，有时会用到dir()，用以将传入的对象属性枚举出来，这里是一个例子：

```
console.log("test", 1, {}, [1,2,3]);
console.dir({one: 1, two: {three: 3}});
```

当你在控制台输入内容时，则不必使用console.log()。为了避免混乱，有些代码片段仍然使用console.log()作输出，并假设所有的代码片段都使用控制台来作检测：

```
window.name === window['name']; // true
```

这和下面这种用法意思一样：

```
console.log(window.name === window['name']);
```

这段代码在控制台中输出为true。

第二章 高质量JavaScript基本要点

本章将对一些实质内容展开讨论，这些内容包括最佳实践、模式和编写高质量JavaScript代码的习惯，比如避免全局变量、使用单var声明、循环中的length预缓存、遵守编码约定等等。本章还包括一些非必要的编程习惯，但更多的关注点将放在总体的代码创建过程上，包括撰写API文档、组织相互评审以及使用JSLint。这些习惯和最佳实践可以帮助你写出更好的、更易读的和可维护的代码，当几个月后或数年后再重读你的代码时，你就会深有体会了。

编写可维护的代码

修复软件bug成本很高，而且随着时间的推移，它们造成的损失也越来越大，特别是在已经打包发布了的软件发现了bug的时候。当然最好是发现bug立刻解决掉，但前提是你对你的代码依然很熟悉，否则当你转身投入到另外一个项目的开发中后，根本不记得当初代码的模样了。过了一段时间后你再去阅读当初的代码你需要：

- 时间来重新学习并理解问题
- 时间去理解问题相关的代码

对大型项目或者公司来说还有一个不得不考虑的问题，就是解决这个bug的人和制造这个bug的人往往不是同一个人。因此减少理解代码所需的时间成本就显得非常重要，不管是隔了很长时间重读自己的代码还是阅读团队内其他人的代码。这对于公司的利益底线和工程师的幸福指数同样重要，因为每个人都宁愿去开发新的项目而不愿花很多时间和精力去维护旧代码。

另外一个软件开发中的普遍现象是，在读代码上花的时间要远远超过写代码的时间。常常当你专注于某个问题的时候，你会坐下来用一下午的时间产出大量的代码。当时的场景下代码是可以正常运行的，但当应用趋于成熟，会有很多因素促使你重读代码、改进代码或对代码做微调。比如：

- 发现了bug
- 需要给应用添加新需求
- 需要将应用迁移到新的平台中运行（比如当市场中出现了新的浏览器时）
- 代码重构
- 由于架构更改或者更换另一种语言导致代码重写

这些不确定因素带来的后果是，少数人花几小时写的代码需要很多人花几个星期去阅读它。因此，创建可维护的代码对于一个成功的应用来说至关重要。

可维护的代码意味着代码是：

- 可读的

- 一致的
- 可预测的
- 看起来像是同一个人写的
- 有文档的

本章接下来的部分会对这几点深入讲解。

减少全局对象

JavaScript 使用函数来管理作用域，在一个函数内定义的变量称作“局部变量”，局部变量在函数外部是不可见的。另一方面，“全局变量”是不在任何函数体内部声明的变量，或者是直接使用而未明的变量。

每一个JavaScript运行环境都有一个“全局对象”，不在任何函数体内使用this就可以获得对这个全局对象的引用。你所创建的每一个全局变量都是这个全局对象的属性。为了方便起见，浏览器都会额外提供一个全局对象的属性window，（常常）用以指向全局对象本身。下面的示例代码中展示了如何在浏览器中创建或访问全局变量：

```
myglobal = "hello"; // antipattern
console.log(myglobal); // "hello"
console.log(window.myglobal); // "hello"
console.log(window["myglobal"]); // "hello"
console.log(this.myglobal); // "hello"
```

全局对象带来的困扰

全局变量的问题是，它们在JavaScript代码执行期间或者整个web页面中始终是可见的。它们存在于同一个命名空间中，因此命名冲突的情况时有发生，毕竟在应用程序的不同模块中，经常会出于某种目的定义相同的全局变量。

同样，常常网页中所嵌入的代码并不是这个网页的开发者所写，比如：

- 网页中使用了第三方的JavaScript库
- 网页中使用了广告代码
- 网页中使用了用以分析流量和点击率的第三方统计代码
- 网页中使用了很多组件，挂件和按钮等等

假设某一段第三方提供的脚本定义了一个全局变量result。随后你在自己写的某个函数中也定义了一个全局变量result。这时，第二个变量就会覆盖第一个，这时就会导致第三方脚本停止工作。

因此，为了让你的脚本和这个页面中的其他脚本和谐相处，要尽可能少的使用全局变量，这一点非常重要。本书随后的章节中会讲到一些减少全局变量的技巧和策略，比如使用命名空间或者立即执行的匿名函数等，但减少全局变量最有效的方法是坚持使用var来声明变量。

由于JavaScript的特点，我们经常有意无意的创建全局变量，毕竟在JavaScript中创建全局变量实在太简单了。首先，你可以不声明而直接使用变量，再者，JavaScirpt中具有“隐式全局对象”的概念，也就是说任何不通过var声明（译注：在JavaScript1.7及以后的版本中，可以通过let来声明块级作用域的变量）的变量都会成为全局对象的一个属性（可以把它们当作全局变量）。看一下下面这段代码：

```
function sum(x, y) {
    // antipattern: implied global
    result = x + y;
    return result;
}
```

这段代码中，我们直接使用了result而没有事先声明它。这段代码是能够正常工作的，但在调用这个方法之后，会产生一个全局变量result，这会带来其他问题。

解决办法是，总是使用var来声明变量，下面代码就是改进了的sum()函数：

```
function sum(x, y) {
    var result = x + y;
    return result;
}
```

这里我们要注意一种反模式，就是在var声明中通过链式赋值的方法创建全局变量。在下面这个代码片段中，a是局部变量，但b是全局变量，而作者的意图显然不是如此：

```
// antipattern, do not use
function foo() {
    var a = b = 0;
    // ...
}
```

为什么会这样？因为这里的计算顺序是从右至左的。首先计算表达式b=0，这里的b是未声明的，这个表达式的值是0，然后通过var创建了局部变量a，并赋值为0。换言之，可以等价的将代码写成这样：

```
var a = (b = 0);
```

如果变量b已经被声明，这种链式赋值的写法是ok的，不会意外的创建全局变量，比如：

```
function foo() {
    var a, b;
    // ...
    a = b = 0; // both local
}
```

> 避免使用全局变量的另一个原因是出于可移植性考虑的，如果你希望将你的代码运行于不同的平台环境（宿主），使用全局变量则非常危险。很有可能你无意间创建的某个全局变量在当前的平台环境中是不存在的，你认为可以安全的使用，而在其他的环境中却是存在的。

忘记var时的副作用

隐式的全局变量和显式定义的全局变量之间有着细微的差别，差别在于通过delete来删除它们的时候表现不一致。

- 通过var创建的全局变量（在任何函数体之外创建的变量）不能被删除。
- 没有用var创建的隐式全局变量（不考虑函数内的情况）可以被删除。

也就是说，隐式全局变量并不算是真正的变量，但他们是全局对象的属性成员。属性是可以通过delete运算符删除的，而变量不可以被删除：

> （译注：在浏览器环境中，所有 JavaScript 代码都是在 window 作用域内的，所以在这种情况下，我们所说的全局变量其实都是 window 下的一个属性，故可以用 delete 删除，但在如 nodejs 或 gjs 等非浏览器环境下，显式声明的全局变量无法用 delete 删除。）

```
// define three globals
var global_var = 1;
global_novar = 2; // antipattern
(function () {
    global_fromfunc = 3; // antipattern
}());

// attempt to delete
delete global_var; // false
delete global_novar; // true
delete global_fromfunc; // true

// test the deletion
typeof global_var; // "number"
typeof global_novar; // "undefined"
typeof global_fromfunc; // "undefined"
```

在ES5严格模式中，给未声明的变量赋值会报错（比如这段代码中提到的两个反模式）。

访问全局对象

在浏览器中，我们可以随时随地通过window属性来访问全局对象（除非你定义了一个名叫window的局部变量）。但换一个运行环境这个方便的window可能就换成了别的名字（甚至根本就被禁止访问全局对象了）。如果不想通过这种写死window的方式来得到全局变量，有一个办法，你可以在任意层次嵌套的函数作用域内执行：

```
var global = (function () {
    return this;
}());
```

这种方式总是可以得到全局对象，因为在被当作函数执行的函数体内（而不是被当作构造函数执行的函数体内），this总是指向全局对象。但这种情况在ECMAScript5的严格模式中行不通，因此在严格模式中你不得不寻求其他的替代方案。比如，如果你在开发一个库，你会将你的代码包装在一个立即执行的匿名函数中（在第四章会讲到），然后从全局作用域中给这个匿名函数传入一个指向this的参数。

单 var 模式

在函数的顶部使用一个单独的var语句是非常推荐的一种模式，它有如下一些好处：

- 在同一个位置可以查找到函数所需的所有变量
- 避免当在变量声明之前使用这个变量时产生的逻辑错误（参照下一小节“声明提前：分散的 var 带来的问题”）
- 提醒你不要忘记声明变量，顺便减少潜在的全局变量
- 代码量更少（输入更少且更易做代码优化）

单var模式看起来像这样：

```
function func() {
    var a = 1,
        b = 2,
        sum = a + b,
        myobject = {},
        i,
        j;
    // function body...
}
```

你可以使用一个var语句来声明多个变量，变量之间用逗号分隔。也可以在这个语句中加入变量的初始化，这是一个非常好的实践。这种方式可以避免逻辑错误（所有未初始化的变量都被声明了，且值为undefined）并增加了代码的可读性。过段时间后再看这段代码，你会体会到声明不同类型变量的惯用名称，比如，你一眼就可看出某个变量是对象还是整数。

你可以在声明变量时多做一些额外的工作，比如在这个例子中就写了sum=a+b这种代码。另一个例子就是当代码中用到对DOM元素时，你可以把对DOM的引用赋值给一些变量，这一步就可以放在一个单独的声明语句中，比如下面这段代码：

```
function updateElement() {
    var el = document.getElementById("result"),
        style = el.style;
    // do something with el and style...
}
```

声明提前：分散的 var 带来的问题

JavaScript 中是允许在函数的任意地方写任意多个var语句的，其实相当于在函数体顶部声明变量，这种现象被称为“变量提前”，当你在声明之前使用这个变量时，可能会造成逻辑错误。对于JavaScript来说，一旦在某个作用域（同一个函数内）里声明了一个变量，这个变量在整个作用域内都是存在的，包括在var声明语句之前。看一下这个例子：

```
// antipattern
myname = "global"; // global variable
function func() {
    alert(myname); // "undefined"
    var myname = "local";
    alert(myname); // "local"
}
func();
```

这个例子中，你可能期望第一个alert()弹出“global”，第二个alert()弹出“local”。这种结果看起来是合乎常理的，因为在第一个alert执行时，myname还没有声明，这时就应该“寻找”全局变量中的myname。但实际情况并不是这样，第一个alert弹出“undefined”，因为myname已经在函数内有声明了（尽管声明语句在后面）。所有的变量声明都提前到了函数的顶部。因此，为了避免类似带有“歧义”的程序逻辑，最好在使用之前一起声明它们。

上一个代码片段等价于下面这个代码片段：

```
myname = "global"; // global variable
function func() {
    var myname; // same as -> var myname = undefined;
    alert(myname); // "undefined"
    myname = "local";
    alert(myname); // "local"
}
func();
```

> 这里有必要对“变量提前”作进一步补充，实际上从JavaScript引擎的工作机制上看，这个过程稍微有点复杂。代码处理经过了两个阶段，第一阶段是创建变量、函数和参数，这一步是预编译的过程，它会扫描整段代码的上下文。第二阶段是代码的运行，这一阶段将创建函数表达式和一些非法的标识符（未声明的变量）。从实用性角度来讲，我们更愿意将这两个阶段归成一个概念“变量提前”，尽管这个概念并没有在ECMAScript标准中定义，但我们常常用它来解释预编译的行为过程。

for 循环

在for循环中，可以对数组或类似数组的对象（比如arguments和HTMLCollection对象）作遍历，最普通的for循环模式形如：

```
// sub-optimal loop
for (var i = 0; i < myarray.length; i++) {
    // do something with myarray[i]
}
```

这种模式的问题是，每次遍历都会访问数组的length属性。这降低了代码运行效率，特别是当myarray并不是一个数组而是一个HTMLCollection对象的时候。

HTMLCollection是由DOM方法返回的对象，比如：

- document.getElementsByName()
- document.getElementsByClassName()
- document.getElementsByTagName()

还有很多其他的HTMLCollection，这些对象是在DOM标准之前就已经在用了，这些HTMLCollection主要包括：

document.images

页面中所有的IMG元素

document.links

页面中所有的A元素

document.forms

页面中所有的表单

document.forms[0].elements

页面中第一个表单的所有字段

这些对象的问题在于，它们均是指向文档（HTML页面）中的活动对象。也就是说每次通过它们访问集合的length时，总是会去查询DOM，而DOM操作则是很耗资源的。

更好的办法是为for循环缓存住要遍历的数组的长度，比如下面这段代码：

```
for (var i = 0, max = myarray.length; i < max; i++) {
    // do something with myarray[i]
}
```

通过这种方法只需要访问DOM节点一次以获得length，在整个循环过程中就都可以使用它。

不管在什么浏览器中，在遍历HTMLCollection时缓存length都可以让程序执行的更快，可以提速两倍（Safari3）到一百九十倍（IE7）不等。更多细节可以参照Nicholas Zakas的《高性能JavaScript》，这本书也是由O'Reilly出版。

需要注意的是，当你在循环过程中需要修改这个元素集合（比如增加DOM元素）时，你更希望更新length而不是更新常量。

遵照单var模式，你可以将var提到循环的外部，比如：

```
function looper() {
    var i = 0,
        max,
        myarray = [];
    // ...
    for (i = 0, max = myarray.length; i < max; i++) {
        // do something with myarray[i]
    }
}
```

这种模式带来的好处就是提高了代码的一致性，因为你越来越依赖这种单var模式。缺点就是在重构代码的时候不能直接复制粘贴一个循环体，比如，你正在将某个循环从一个函数拷贝至另外一个函数中，必须确保i和max也拷贝至新函数里，并且需要从旧函数中将这些没用的变量删除掉。

最后一个需要对循环做出调整的地方是将i++替换成为下面两者之一：

```
i = i + 1
i += 1
```

JSLint提示你这样做，是因为++和--实际上降低了代码的可读性，如果你觉得无所谓，可以将JSLint的plusplus选项设为false（默认为true），本书所介绍的最后一个模式用到了: i += 1。

关于这种for模式还有两种变化的形式，做了少量改进，原因有二：

- 减少一个变量（没有max）
- 减量循环至0，这种方式速度更快，因为和零比较要比和非零数字或数组长度比较要高效的多

第一种变化形式是：

```
var i, myarray = [];
for (i = myarray.length; i--;) {
    // do something with myarray[i]
}
```

第二种变化形式用到了while循环：

```
var myarray = [],
    i = myarray.length;
while (i--) {
    // do something with myarray[i]
}
```

这些小改进只体现在性能上，此外，JSLint不推荐使用i--。

for-in 循环

for-in 循环用于对非数组对象作遍历。通过for-in进行循环也被称作“枚举”。

从技术角度讲，for-in循环同样可以用于数组（JavaScript中数组即是对象），但不推荐这样做。当使用自定义函数扩充了数组对象时，这时更容易产生逻辑错误。另外，for-in循环中属性的遍历顺序是不固定的，所以最好数组使用普通的for循环，对象使用for-in循环。

可以使用对象的hasOwnProperty()方法将从原型链中继承来的属性过滤掉，这一点非常重要。看一下这段代码：

```
// the object
var man = {
    hands: 2,
    legs: 2,
    heads: 1
};
// somewhere else in the code
// a method was added to all objects
if (typeof Object.prototype.clone === "undefined") {
    Object.prototype.clone = function () {};
}
```

在这段例子中，我们定义了一个名叫man的对象直接量。在代码中的某个地方（可以是man定义之前也可以是之后），给Object的原型中增加了一个方法clone()。原型链是实时的，这意味着所有的对象都可以访问到这个新方法。要想在枚举man的时候避免枚举出clone()方法，则需要调用hasOwnProperty()来对原型属性进行过滤。如果不做过滤，clone()也会被遍历到，而这不是我们所希望的：

```
// 1.
// for-in loop
for (var i in man) {
    if (man.hasOwnProperty(i)) { // filter
        console.log(i, ":", man[i]);
    }
}
/*
result in the console
hands : 2
legs : 2
heads : 1
*/

// 2.
// antipattern:
// for-in loop without checking hasOwnProperty()
for (var i in man) {
    console.log(i, ":", man[i]);
}
/*
result in the console
hands : 2
legs : 2
heads : 1
clone: function()
*/
```

另外一种的写法是通过Object.prototype直接调用hasOwnProperty()方法，像这样：

```
for (var i in man) {
    if (Object.prototype.hasOwnProperty.call(man, i)) { // filter
        console.log(i, ":", man[i]);
    }
}
```

这种做法的好处是，当man对象中重新定义了hasOwnProperty方法时，可以避免调用时的命名冲突（译注：明确指定调用的是Object.prototype上的方法而不是实例对象中的方法），这种做法同样可以避免冗长的属性查找过程（译注：这种查找过程多是在原型链上进行查找），一直查找到Object中的方法，你可以定义一个变量来“缓存”住它（译注：这里所指的是缓存住Object.prototype.hasOwnProperty）：

```
var i,
    hasOwn = Object.prototype.hasOwnProperty;
for (i in man) {
    if (hasOwn.call(man, i)) { // filter
        console.log(i, ":", man[i]);
    }
}
```

> 严格说来，省略hasOwnProperty()并不是一个错误。根据具体的任务以及你对代码的自信程度，你可以省略掉它以提高一些程序执行效率。但当你对当前要遍历的对象不确定的时候，添加hasOwnProperty()则更加保险些。

这里提到一种格式上的变化写法（这种写法无法通过JSLint检查），这种写法在for循环所在的行加入了if判断条件，他的好处是能让循环语句读起来更完整和通顺（“如果元素包含属性X，则拿X做点什么”）：

```
// Warning: doesn't pass JSLint
var i,
    hasOwn = Object.prototype.hasOwnProperty;
for (i in man) if (hasOwn.call(man, i)) { // filter
    console.log(i, ":", man[i]);
}
```

（不）扩充内置原型

我们可以扩充构造函数的prototype属性，这是一种非常强大的特性，用来为构造函数增加功能，但有时这个功能强大到超过我们的掌控。

给内置构造函数比如Object()、Array()、和Function()扩充原型看起来非常诱人，但这种做法严重降低了代码的可维护性，因为它让你的代码变得难以预测。对于那些基于你的代码做开发的开发者来说，他们更希望使用原生的JavaScript方法来保持工作的连续性，而不是使用你所添加的方法（译注：因为原生的方法更可靠，而你写的方法可能会有bug）。

另外，如果将属性添加至原型中，很可能导致在那些不使用hasOwnProperty()做检测的循环中将原型上的属性遍历出来，这会造成混乱。

因此，不扩充内置对象的原型是最好的，你也可以自己定义一个规则，仅当下列条件满足时做例外考虑：

1. 未来的ECMAScript版本的JavaSciprt会将你实现的方法添加为内置方法。比如，你可以实现ECMAScript5定义的一些方法，一直等到浏览器升级至支持ES5。这样，你只是提前定义了这些有用的方法。
2. 如果你发现你自定义的方法已经不存在，要么已经在代码其他地方实现了，要么是浏览器的JavaScript引擎已经内置实现了。
3. 你所做的扩充附带充分的文档说明，且和团队其他成员做了沟通。

如果你遇到这三种情况之一，你可以给内置原型添加自定义方法，写法如下：

```
if (typeof Object.protoype.myMethod !== "function") {
    Object.protoype.myMethod = function () {
        // implementation...
    };
}
```

switch 模式

你可以通过下面这种模式的写法来增强switch语句的可读性和健壮性：

```
var inspect_me = 0,
    result = '';
switch (inspect_me) {
case 0:
    result = "zero";
    break;
case 1:
    result = "one";
    break;
default:
    result = "unknown";
}
```

这个简单的例子所遵循的风格约定如下：

- 每个case和switch对齐（这里不考虑花括号相关的缩进规则）
- 每个case中的代码整齐缩进
- 每个case都以break作为结束
- 避免连续执行多个case语句块（当省略break时会发生），如果你坚持认为连续执行多case语句块是最好的方法，请务必补充文档说明，对于其他人来说，这种情况看起来是错误的。
- 以default结束整个switch，以确保即便是在找不到匹配项时也会有正常的结果，

避免隐式类型转换

在JavaScript的比较操作中会有一些隐式的数据类型转换。比如诸如false == 0或""==0之类的比较都返回true。

为了避免隐式类型转换造对程序造成干扰，推荐使用===和!===运算符，它们较除了比较值还会比较类型。

```
var zero = 0;
if (zero === false) {
    // not executing because zero is 0, not false
}
// antipattern
if (zero == false) {
    // this block is executed...
}
```

另外一种观点认为当==够用的时候就不必多余的使用===。比如，当你知道typeof的返回值是一个字符串，就不必使用全等运算符。但JSLint却要求使用全等运算符，这当然会提高代码风格的一致性，并减少了阅读代码时的思考（“这里使用==是故意的还是无意的？”）。

避免使用eval()

当你想使用eval()的时候，不要忘了那句话“eval()是魔鬼”。这个函数的参数是一个字符串，它可以执行任意字符串。如果事先知道要执行的代码是有问题的（在运行之前），则没有理由使用eval()。如果需要在运行时动态生成执行代码，往往都会有更佳的方式达到同样的目的，而非一定要使用eval()。例如，访问动态属性时可以使用方括号：

```
// antipattern
var property = "name";
alert(eval("obj." + property));
// preferred
var property = "name";
alert(obj[property]);
```

eval()同样有安全隐患，因为你需要运行一些容易被干扰的代码（比如运行一段来自于网络的代码）。在处理Ajax请求所返回的JSON数据时会常遇到这种情况，使用eval()是一种反模式。这种情况下最好使用浏览器的内置方法来解析JSON数据，以确保代码的安全性和数据的合法性。如果浏览器不支持JSON.parse()，你可以使用JSON.org所提供的库。

记住，多数情况下，给setInterval()、setTimeout()和Function()构造函数传入字符串的情形和eval()类似，这种用法也是应当避免的，这一点非常重要，因为这些情形中JavaScript最终还是会执行传入的字符串参数：

```
// antipatterns
setTimeout("myFunc()", 1000);
setTimeout("myFunc(1, 2, 3)", 1000);
// preferred
setTimeout(myFunc, 1000);
setTimeout(function () {
    myFunc(1, 2, 3);
}, 1000);
```

new Function()的用法和eval()非常类似，应当特别注意。这种构造函数的方式很强大，但往往被误用。如果你不得不使用eval()，你可以尝试用new Function()来代替。这有一个潜在的好处，在new Function()中运行的代码会在一个局部函数作用域内执行，因此源码中所有用var定义的变量不会自动变成全局变量。还有一种方法可以避免eval()中定义的变量转换为全局变量，即是将eval()包装在一个立即执行的匿名函数内（详细内容请参照第四章）。

看一下这个例子，这里只有un成为了全局变量，污染了全局命名空间：

```
console.log(typeof un);// "undefined"
console.log(typeof deux); // "undefined"
console.log(typeof trois); // "undefined"

var jsstring = "var un = 1; console.log(un);";
eval(jsstring); // logs "1"

jsstring = "var deux = 2; console.log(deux);";
new Function(jsstring)(); // logs "2"

jsstring = "var trois = 3; console.log(trois);";
(function () {
    eval(jsstring);
}()); // logs "3"

console.log(typeof un); // "number"
console.log(typeof deux); // "undefined"
console.log(typeof trois); // "undefined"
```

eval()和Function构造函数还有一个区别，就是eval()可以修改作用域链，而Function更像是一个沙箱。不管在什么地方执行Function，它只能看到全局作用域。因此它不会太严重的污染局部变量。在下面的示例代码中，eval()可以访问且修改其作用域之外的变量，而Function不能（注意，使用Function和new Function是完全一样的）。

```
(function () {
    var local = 1;
    eval("local = 3; console.log(local)"); // logs 3
    console.log(local); // logs 3
}());

(function () {
    var local = 1;
    Function("console.log(typeof local);")(); // logs undefined
}());
```

使用parseInt()进行数字转换

可以使用parseInt()将字符串转换为数字。函数的第二个参数是转换基数（译注：“基数”指的是数字进制的方式），这个参数通常被省略。但当字符串以0为前缀时转换就会出错，例如，在表单中输入日期的一个字段。ECMAScript3中以0为前缀的字符串会被当作八进制数处理（基数为8）。但在ES5中不是这样。为了避免转换类型不一致而导致的意外结果，应当总是指定第二个参数：

```
var month = "06",
    year = "09";
month = parseInt(month, 10);
year = parseInt(year, 10);
```

在这个例子中，如果省略掉parseInt的第二个参数，比如parseInt(year)，返回值是0，因为“09”被认为是八进制数（等价于parseInt(year,8)），而且09是非法的八进制数。

字符串转换为数字还有两种方法：

```
+"08" // result is 8
Number("08") // 8
```

这两种方法要比parseInt()更快一些，因为顾名思义parseInt()是一种“解析”而不是简单的“转换”。但当你期望将“08 hello”这类字符串转换为数字，则必须使用parseInt()，其他方法都会返回NaN。

编码风格

确立并遵守编码规范非常重要，这会让你的代码风格一致、可预测、可读性更强。团队新成员通过学习编码规范可以很快进入开发状态、并写出团队其他成员易于理解的代码。

在开源社区和邮件组中关于编码风格的争论一直不断（比如关于代码缩进，用tab还是空格？）。因此，如果你打算在团队内推行某种编码规范时，要做好应对各种反对意见的心理准备，而且要吸取各种意见，这对确立并一贯遵守某种编码规范是非常重要，而不是斤斤计较的纠结于编码规范的细节。

缩进

代码没有缩进几乎就不能读了，而不一致的缩进更加糟糕，因为它看上去像是遵循了规范，真正读起来却磕磕绊绊。因此规范的使用缩进非常重要。

有些开发者喜欢使用tab缩进，因为每个人都可以根据自己的喜好来调整tab缩进的空格数，有些人则喜欢使用空格缩进，通常是四个空格，这都无所谓，只要团队每个人都遵守同一个规范即可，本书中所有的示例代码都采用四个空格的缩进写法，这也是JSLint所推荐的。

那么到底什么应该缩进呢？规则很简单，花括号里的内容应当缩进，包括函数体、循环（do、while、for和for-in）体、if条件、switch语句和对象直接量里的属性。下面的代码展示了如何正确的使用缩进：

```
function outer(a, b) {
    var c = 1,
        d = 2,
        inner;
    if (a > b) {
        inner = function () {
            return {
                r: c - d
            };
        };
    } else {
        inner = function () {
            return {
                r: c + d
            };
        };
    }
    return inner;
}
```

花括号

应当总是使用花括号，即使是在可省略花括号的时候也应当如此。从技术角度讲，如果if或for中只有一个语句，花括号是可以省略的，但最好还是不要省略。这让你的代码更加工整一致而且易于更新。

假设有这样一段代码，for循环中只有一条语句，你可以省略掉这里的花括号，而且不会有语法错误：

```
// bad practice
for (var i = 0; i < 10; i += 1)
    alert(i);
```

但如果过了一段时间，你给这个循环添加了另一行代码？

```
// bad practice
for (var i = 0; i < 10; i += 1)
    alert(i);
    alert(i + " is " + (i % 2 ? "odd" : "even"));
```

第二个alert实际处于循环体之外，但这里的缩进会迷惑你。长远考虑最好还是写上花括号，即便是在只有一个语句的语句块中也应如此：

```
// better
for (var i = 0; i < 10; i += 1) {
    alert(i);
}
```

同理，if条件句也应当如此：

```
// bad
if (true)
    alert(1);
else
    alert(2);

// better
if (true) {
    alert(1);
} else {
    alert(2);
}
```

左花括号的位置

开发人员对于左大括号的位置有着不同的偏好，在同一行呢还是在下一行？

```
if (true) {
    alert("It's TRUE!");
}
```

或者：

```
if (true)
{
    alert("It's TRUE!");
}
```

在这个例子中，看起来只是个人偏好问题。但有时候花括号位置的不同则会影响程序的执行。因为JavaScript会“自动插入分号”。JavaScript对行结束时的分号并无要求，它会自动将分号补全。因此，当函数return语句返回了一个对象直接量，而对象的左花括号和return不在同一行时，程序的执行就和预想的不同了：

```
// warning: unexpected return value
function func() {
    return
    {
        name: "Batman"
    };
}
```

可以看出程序作者的意图是返回一个包含了name属性的对象，但实际情况不是这样。因为return后会填补一个分号，函数的返回值就是undefined。这段代码等价于：

```
// warning: unexpected return value
function func() {
    return undefined;
    // unreachable code follows...
    {
        name: "Batman"
    };
}
```

结论，总是使用花括号，而且总是将左花括号与上一条语句放在同一行：

```
function func() {
    return {
        name: "Batman"
    };
}
```

> 关于分号应当注意：和花括号一样，应当总是使用分号，尽管在JavaScript解析代码时会补全行末省略的分号。严格遵守这条规则，可以让代码更加严谨，同时可以避免前面例子中所出现的歧义。

空格

空格的使用同样有助于改善代码的可读性和一致性。在写英文句子的时候，在逗号和句号后面会使用间隔。在JavaScript中，你可以按照同样的逻辑在表达式（相当于逗号）和语句结束（相对于完成了某个"想法"）后面添加间隔。

适合使用空格的地方包括：

- for循环中的分号之后，比如 `for (var i = 0; i < 10; i += 1) {...}`
- for循环中初始化多个变量，比如 `for (var i = 0, max = 10; i < max; i += 1) {...}`
- 分隔数组项的逗号之后，`var a = [1, 2, 3];`
- 对象属性后的逗号以及名值对之间的冒号之后，`var o = {a: 1, b: 2};`
- 函数参数中，`myFunc(a, b, c)`
- 函数声明的花括号之前，`function myFunc() {}`
- 匿名函数表达式function之后，`var myFunc = function () {};`

另外，我们推荐在运算符和操作数之间添加空格。也就是说在+, -, *, =, <, >, <=, >=, ===, !==, &&, ||, +=符号前后都添加空格。

```
// generous and consistent spacing
// makes the code easier to read
// allowing it to "breathe"
var d = 0,
    a = b + 1;
if (a && b && c) {
    d = a % c;
    a += d;
}

// antipattern
// missing or inconsistent spaces
// make the code confusing
var d= 0,
    a =b+1;
if (a&& b&&c) {
    d=a %c;
    a+= d;
}
```

最后，还应当注意，最好在花括号旁边添加空格：

- 在函数、if-else语句、循环、对象直接量的左花括号之前补充空格（{）
- 在右花括号和else和while之间补充空格

> 垂直空白的使用经常被我们忽略，你可以使用空行来将代码单元分隔开，就像文学作品中使用段落作分隔一样。

命名规范

另外一种可以提升你代码的可预测性和可维护性的方法是采用命名规范。也就是说变量和函数的命名都遵照同种习惯。

下面是一些建议的命名规范，你可以原样采用，也可以根据自己的喜好作调整。同样，遵循规范要比规范本身更加重要。

构造器命名中的大小写

JavaScript中没有类，但有构造函数，可以通过new来调用构造函数：

```
var adam = new Person();
```

由于构造函数毕竟还是函数，不管我们将它用作构造器还是函数，当然希望只通过函数名就可分辨出它是构造器还是普通函数。

首字母大写可以提示你这是一个构造函数，而首字母小写的函数一般只认为它是普通的函数，不应该通过new来调用它：

```
function MyConstructor() {...}
function myFunction() {...}
```

下一章将介绍一些强制将函数用作构造器的编程模式，但遵守我们所提到的命名规范会更好的帮助程序员阅读源码。

单词分隔

当你的变量名或函数名中含有多个单词时，单词之间的分隔也应当遵循统一的约定。最常见的做法是“驼峰式”命名，单词都是小写，每个单词的首字母是大写。

对于构造函数，可以使用“大驼峰式”命名，比如MyConstructor()，对于函数和方法，可以采用“小驼峰式”命名，比如myFunction()，calculateArea()和getFirstName()。

那么对于那些不是函数的变量应当如何命名呢？变量名通常采用小驼峰式命名，还有一个不错的做法是，变量所有字母都是小写，单词之间用下划线分隔，比如，first_name，favorite_bands和old_company_name，这种方法可以帮助你区分函数和其他标识符——原始数据类型或对象。

ECMAScript的属性和方法均使用Camel标记法，尽管多字的属性名称是罕见的（正则表达式对象的lastIndex和ignoreCase属性）。

在ECMAScript中的属性和方法均使用驼峰式命名，尽管包含多单词的属性名称（正则表达式对象中的lastIndex和ignoreCase）并不常见。

其他命名风格

有时开发人员使用命名规范来弥补或代替语言特性的不足。

比如，JavaScript中无法定义常量（尽管有一些内置常量比如Number.MAX_VALUE），所以开发者都采用了这种命名习惯，对于那些程序运行周期内不会更改的变量使用全大写字母来命名。比如：

```
// precious constants, please don't touch
var PI = 3.14,
    MAX_WIDTH = 800;
```

除了使用大写字母的命名方式之外，还有另一种命名规约：全局变量都大写。这种命名方式和“减少全局变量”的约定相辅相成，并让全局变量很容易辨认。

除了常量和全局变量的命名惯例，这里讨论另外一种命名惯例，即私有变量的命名。尽管在JavaScript是可以实现真正的私有变量的，但开发人员更喜欢在私有成员或方法名之前加上下划线前缀，比如下面的例子：

```
var person = {
    getName: function () {
        return this._getFirst() + ' ' + this._getLast();
    },
    _getFirst: function () {
        // ...
    },
    _getLast: function () {
        // ...
    }
};
```

在这个例子中，getName()的身份是一个公有方法，属于稳定的API，而_getFirst()和_getLast()则是私有方法。尽管这两个方法本质上和公有方法无异，但在方法名前加下划线前缀就是为了警告用户不要直接使用这两个私有方法，因为不能保证它们在下一个版本中还能正常工作。JSLint会对私有方法作检查，除非设置了JSLint的nomen选项为false。

下面介绍一些_private风格写法的变种：

- 在名字尾部添加下划下以表明私有，比如 `name_` 和 `getElements_()`
- 使用一个下划线前缀表明受保护的属性_protected，用两个下划线前缀表明私有属性__private
- 在Firefox中实现了一些非标准的内置属性，这些属性在开头和结束都有两个下划线，比如 `__proto__` 和 `__parent__`

书写注释

写代码就要写注释，即便你认为你的代码不会被别人读到。当你对一个问题非常熟悉时，你会很快找到问题代码，但当过了几个星期后再来读这段代码，则需要绞尽脑汁的回想代码的逻辑。

你不必对显而易见的代码作过多的注释：每个变量和每一行都作注释。但你需要对所有的函数、他们的参数和返回值补充注释，对于那些有趣的或怪异的算法和技术也应当配备注释。对于阅读你的代码的其他人来说，注释就是一种提示，只要阅读注释、函数名以及参数，就算不读代码也能大概理解程序的逻辑。比如，这里有五到六行代码完成了某个功能，如果提供了一行描述这段代码功能的注释，读程序的人就不必再去关注代码的细节实现了。代码注释的写法并没有硬性规定，有些代码片段（比如正则表达式）的确需要比代码本身还多的注释。

> 由于过时的注释会带来很多误导，这比不写注释还糟糕。因此保持注释时刻更新的习惯非常重要，尽管对很多人来说这很难做到。

在下一小节我们会讲到，注释可以自动生成文档。

书写API文档

很多人都觉得写文档是一件枯燥且吃力不讨好的事情，但实际情况不是这样。我们可以通过代码注释自动生成文档，这样就不用再去专门写文档了。很多人觉得这是一个不错的点子，因为根据某些关键字和格式化的文档自动生成可阅读的参考手册本身就是“某种编程”。

传统的APIdoc诞生自Java世界，这个工具名叫“javadoc”，和Java SDK（软件开发工具包）一起提供。但这个创意迅速被其他语言借鉴。JavaScript领域有两个非常优秀的开源工具，它们是JSDoc Toolkit（http://code.google.com/p/jsdoc-toolkit/ ）和YUIDoc（http://yuilibrary.com/projects/yuidoc ）。

生成API文档的过程包括：

- 以特定的格式来组织书写源代码
- 运行工具来对代码和注释进行解析
- 发布工具运行的结果，通常是HTML页面

你需要学习这种特殊的语法，包括十几种标签，写法类似于：

```
/**
 * @tag value
 */
```

比如这里有一个函数reverse()，可以对字符串进行反序操作。它的参数和返回值都是字符串。给它补充注释如下：

```
/**
* Reverse a string
*
* @param {String} input String to reverse
* @return {String} The reversed string
*/
var reverse = function (input) {
    // ...
    return output;
};
```

可以看到，@param是用来说明输入参数的标签，@return是用来说明返回值的标签，文档生成工具最终会为将这种带注释的源代码解析成格式化好的HTML文档。

一个例子：YUIDoc

YUIDoc最初的目的是为YUI库（Yahoo! User Interface）生成文档，但也可以应用于任何项目，为了更充分的使用YUIDoc你需要学习它的注释规范，比如模块和类的写法（当然在JavaScript中是没有类的概念的）。

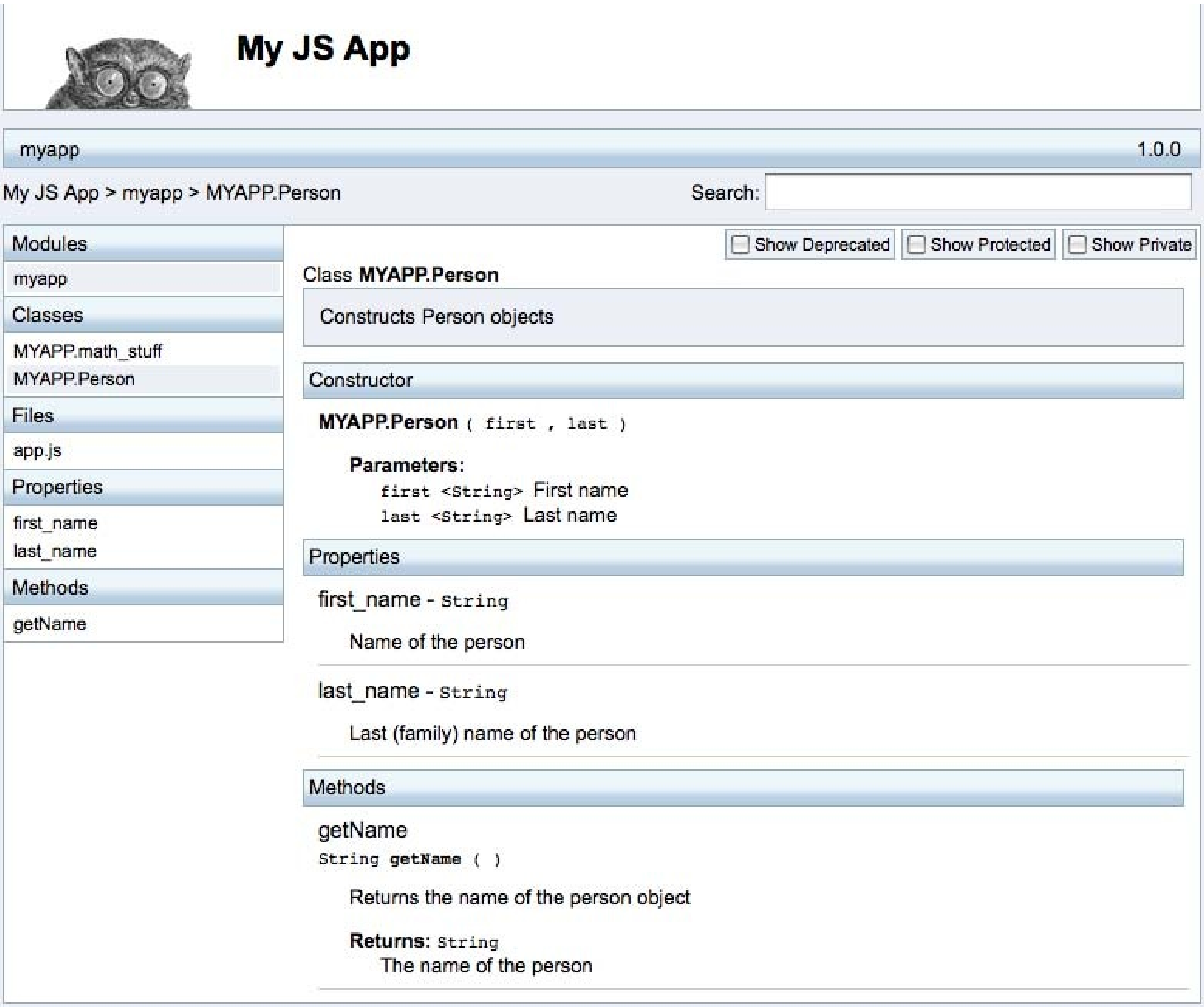
让我们看一个用YUIDoc生成文档的完整例子。

图2-1展示了最终生成的文档的模样，你可以根据项目需要随意定制HTML模板，让生成的文档更加友好和个性化。

这里同样提供了在线的demo，请参照 http://jspatterns.com/book/2/ 。

这个例子中所有的应用作为一个模块（myapp）放在一个文件里（app.js），后续的章节会更详细的介绍模块，现在只需知道用可以用一个YUIDoc的标签来表示模块即可。

图2-1 YUIDoc生成的文档



app.js的开始部分：

```
/**
 * My JavaScript application
 *
 * @module myapp
 */
```

然后定义了一个空对象作为模块的命名空间：

```
var MYAPP = {};
```

紧接着定义了一个包含两个方法的对象math_stuff，这两个方法分别是sum()和multi()：

```
/**
* A math utility
* @namespace MYAPP
* @class math_stuff
*/
MYAPP.math_stuff = {
    /**
    * Sums two numbers
    *
    * @method sum
    * @param {Number} a First number
    * @param {Number} b The second number
    * @return {Number} The sum of the two inputs
    */
    sum: function (a, b) {
        return a + b;
    },

    /**
    * Multiplies two numbers
    *
    * @method multi
    * @param {Number} a First number
    * @param {Number} b The second number
    * @return {Number} The two inputs multiplied
    */
    multi: function (a, b) {
        return a * b;
    }
};
```

这样就结束了第一个“类”的定义，注意粗体表示的标签。

@namespace

指向你的对象的全局引用

@class

代表一个对象或构造函数的不恰当的称谓（JavaScript中没有类）

@method

定义对象的方法，并指定方法的名称

@param

列出函数需要的参数，参数的类型放在一对花括号内，跟随其后的是参数名和描述

@return

和@param类似，用以描述方法的返回值，可以不带名字

我们用构造函数来实现第二个“类”，给这个类的原型添加一个方法，能够体会到YUIDoc采用了不同的方式来创建对象：

```
/**
* Constructs Person objects
* @class Person
* @constructor
* @namespace MYAPP
* @param {String} first First name
* @param {String} last Last name
*/
MYAPP.Person = function (first, last) {
    /**
    * Name of the person
    * @property first_name
    * @type String
    */
    this.first_name = first;
    /**
    * Last (family) name of the person
    * @property last_name
    * @type String
    */
    this.last_name = last;
};
/**
* Returns the name of the person object
*
* @method getName
* @return {String} The name of the person
*/
MYAPP.Person.prototype.getName = function () {
    return this.first_name + ' ' + this.last_name;
};
```

在图2-1中可以看到生成的文档中Person构造函数的生成结果，粗体的部分是：

- @constructor 暗示了这个“类”其实是一个构造函数
- @prototype 和 @type 用来描述对象的属性

YUIDoc工具是语言无关的，只解析注释块，而不是JavaScript代码。它的缺点是必须要在注释中指定属性、参数和方法的名字，比如，@property first_name。好处是一旦你熟练掌握YUIDoc，就可以用它对任何语言源码进行注释的文档化。

编写易读的代码

这种将APIDoc格式的代码注释解析成API参考文档的做法看起来很偷懒，但还有另外一个目的，通过代码重审来提高代码质量。

很多作者或编辑会告诉你“编辑非常重要”，甚至是写一本好书或好文章最最重要的步骤。将想法落实在纸上形成草稿只是第一步，草稿给读者提的信息往往重点不明晰、结构不合理、或不符合循序渐进的阅读习惯。

对于编程也是同样的道理，当你坐下来解决一个问题的时候，这时的解决方案只是一种“草案”，尽管能正常工作，但是不是最优的方法呢？是不是可读性好、易于理解、可维护佳或容易更新？当一段时间后再来review你的代码，一定会发现很多需要改进的地方，需要重新组织

代码或删掉多余的内容等等。这实际上就是在“整理”你的代码了，可以很大程度提高你的代码质量。但事情往往不是这样，我们常常承受着高强度的工作压力，根本没有时间来整理代码，因此通过代码注释写文档其实是不错的机会。

往往在写注释文档的时候，你会发现很多问题。你也会重新思考源代码中不合理之处，比如，某个方法中的第三个参数比第二个参数更常用，第二个参数多数情况下取值为true，因此就需要对这个方法接口进行适当的改造和包装。

写出易读的代码（或API），是指别人能轻易读懂程序的思路。所以你需要采用更好的思路来解决手头的问题。

尽管我们认为“草稿”不甚完美，但至少也算“抱佛脚”的权宜之计，一眼看上去是有点“草”，不过也无所谓，特别是当你处理的是一个关键项目时（会有人命悬与此）。其实你应当扔掉你所给出的第一个解决方案，虽然它是可以正常工作的，但毕竟是一个草率的方案，不是最佳方案。你给出的第二个方案会更加靠谱，因为这时你对问题的理解更加透彻。第二个方案不是简单的复制粘贴之前的代码，也不能投机取巧寻找某种捷径。

相互评审

另外一种可以提高代码质量的方法是组织相互评审。同行的评审很正式也很规范，即便是求助于特定的工具，也不失是一种开发生产线上值得提倡的步骤。但你可能觉得没有时间去作代码互审，没关系，你可以让坐在你旁边的同事读一下你的代码，或者和她（译注：注意是“她”而不是“他”）一起过一遍你的代码。

同样，当你在写APIDoc或任何其他文档的时候，同行的评审能帮助你的产出物更加清晰，因为你写的文档是让别人读的，你必须确保别人能理解你所作的东西。

同行的评审是一种非常不错的习惯，不仅仅是因为它能让代码变得更好，更重要的，在评审的过程中，评审人和代码作者通过分享和讨论，两人都能取长补短、相互促进。

如果你的团队只有你一个开发人员，找不出第二个人能给你作代码评审，这也没关系。你可以通过将你的代码片段开源，或把有意思的代码片段贴在博客中，会有人对你的代码感兴趣的。

另外一个非常好的习惯是使用版本管理工具（CVS，SVN或Git），一旦有人修改并提交了代码，都会发邮件通知组内成员。虽然大部分邮件都进入了垃圾箱，但总是会碰巧有人在工作间隙看到你所提交的代码，并对代码做出一些评价。

生产环境中的代码压缩（**Minify**）

这里所说的代码压缩（Minify）是指去除JavaScript代码中的空格、注释以及其他不必要的部分，用以减少JavaScript文件的体积，降低网络带宽损耗。我们通常使用类似YUICompressor（Yahoo!）或Closure Compiler（Google）的压缩工具来为网页加载提速。

对于生产环境（译注："生产环境"指的是项目上线后的正式环境）中的脚本是需要作压缩的，压缩后的文件体积能减少至原来的一半以下。

下面这段代码是压缩后的样子（这段代码是YUI2库中的Event模块）：

```
YAHOO.util.CustomEvent=function(D,C,B,A){this.type=D;this.scope=C||window;this.silent
=B;this.signature=A||YAHOO.util.CustomEvent.LIST;this.subscribers=[];if(!this.silent)
{}var E="_YUICEOnSubscribe";if(D!==E){this.subscribeEvent=new
YAHOO.util.CustomEvent(E,this,true);}...
```

除了去除空格、空行和注释之外，压缩工具还能缩短命名的长度（前提是保证代码的安全），比如这段代码中的参数A、B、C、D。压缩工具只会重命名局部变量，因为更改全局变量会破坏代码的逻辑。这也是要尽量使用局部变量的原因。如果你使用的全局变量是对DOM节点的引用，而且程序中多次用到，最好将它赋值给一个局部变量，这样能提高查找速度，代码也会运行的更快，此外还能提高压缩比、加快下载速度（译注：在服务器开启Gzip的情况下，对下载速度的影响几乎可以忽略不计）。

补充说明一下，Goolge Closure Compiler还会对全局变量进行压缩（在"高级"模式中），这是很危险的，且对编程规范的要求非常苛刻。它的好处是压缩比非常高。

对生产环境的脚本做压缩是相当重要的步骤，它能提升页面性能，你应当使用工具来完成压缩。千万不要试图手写"压缩好的"代码，你应当坚持使用语义化的变量命名，并保留足够的空格、缩进和注释。你写的代码是需要被人阅读的，所以应当将注意力放在代码可读性和可维护性上，代码压缩的工作交给工具去完成。

运行 JSLint

在上一章我们已经介绍了JSLint，这里我们介绍更多的使用场景。对你的代码进行JSLint检查是非常好的编程习惯，你应该相信这一点。

JSLint的检查点都有哪些呢？它会对本章讨论过的一些模式（单var模式、parseInt()的第二个参数、总是使用花括号）做检查。JSLint还包括其他方面的检查：

- 不可达代码
- 在使用变量之前需要声明
- 不安全的UTF字符
- 使用void、with、和eval
- 无法正确解析的正则表达式

JSLint是基于JavaScript实现的（它是可以通过JSLint检查的），它提供了在线工具，也可以下载使用，可以运行于很多种平台的JavaScript解析器。你可以将源码下载后在本地运行，支持的环境包括WSH（Windows Scripting Host，Windows）、JSC（JavaScriptCore，MacOSX）或Rhino（Mozilla开发的JavaScript引擎）。

可以将JSLint下载后和你的代码编辑器配置在一起，着是一个不错的注意，这样每次你保存代码的时候都会自动执行代码检查（比如配置快捷键）。

小结

本章我们讲解了编写可维护性代码的含义，本章的讨论非常重要，它不仅关系着软件项目的成功与否，还关系到参与项目的工程师的“精神健康”和“幸福指数”。随后我们讨论了一些最佳实践和模式，它们包括：

- 减少全局对象，最好每个应用只有一个全局对象
- 函数都使用单var模式来定义，这样可以将所有的变量放在同一个地方声明，同时可以避免“声明提前”给程序逻辑带来的影响。
- for循环、for-in循环、switch语句、“禁止使用eval()”、不要扩充内置原型
- 遵守统一的编码规范（在任何必要的时候保持空格、缩进、花括号和分号）和命名约定（构造函数、普通函数和变量）。

本章还讨论了其他一些和代码本身无关的实践，这些实践和编码过程紧密相关，包括书写注释、生成API文档，组织代码评审、不要试图去手动了“压缩”（minify）代码而牺牲代码可读性、坚持使用JSLint来对代码做检查。

第三章 直接量和构造函数

JavaScript中的直接量模式更加简洁、富有表现力，且在定义对象时不容易出错。本章将对直接量展开讨论，包括对象、数组和正则表达式直接量，以及为什么要优先使用它们而不是如 `Object()` 和 `Array()` 这些等价的内置构造器函数。本章同样会介绍JSON格式，JSON是使用数组和对象直接量的形式定义的一种数据转换格式。本章还会讨论自定义构造函数，包括如何强制使用new以确保构造函数的正确执行。

本章还会补充讲述一些基础知识，比如内置包装对象Number()、String()和Boolean()，以及如何将它们和原始值（数字、字符串和布尔值）比较。最后，快速介绍一下Error()构造函数的用法。

对象直接量

我们可以将JavaScript中的对象简单的理解为名值对组成的散列表（hash table），在其他编程语言中被称作“关联数组”。其中的值可以是原始值也可以是对象，不管是什么类型，它们都是“属性”（properties），属性值同样可以是函数，这时属性就被称为“方法”（methods）。

JavaScript中自定义的对象（用户定义的本地对象）任何时候都是可变的。内置本地对象的属性也是可变的。你可以先创建一个空对象，然后在需要时给它添加功能。“对象直接量写法（object literal notation）”是按需创建对象的一种理想方式。

看一下这个例子：

```
// start with an empty object
var dog = {};

// add one property
dog.name = "Benji";

// now add a method
dog.getName = function () {
    return dog.name;
};
```

在这个例子中，我们首先定义了一个空对象，然后添加了一个属性和一个方法，在程序的生命周期内的任何时刻都可以：

1.更改属性和方法的值，比如：

```
dog.getName = function () {
    // redefine the method to return
    // a hardcoded value
    return "Fido";
};
```

2.完全删除属性/方法

```
delete dog.name;
```

3.添加更多的属性和方法

```
dog.say = function () {
    return "Woof!";
};
dog.fleas = true;
```

其实不必每次开始都创建空对象，对象直接量模式可以直接在创建对象时添加功能，就像下面这个例子所展示的：

```
var dog = {
    name: "Benji",
    getName: function () {
        return this.name;
    }
};
```

> 在本书中多次提到“空对象”（“blank object”和“empty object”）。这只是某种简称，要知道JavaScript中根本不存在真正的空对象，理解这一点至关重要。即使最简单的`{}`对象包含从Object.prototype继承来的属性和方法。我们提到的“空（empty）对象”只是说这个对象没有自己的属性，不考虑它是否有继承来的属性。

对象直接量语法

如果你从来没有接触过对象直接量写法，第一次碰到可能会感觉怪怪的。但越到后来你就越喜欢它。本质上讲，对象直接量语法包括：

- 将对象主体包含在一对花括号内（`{` and `}`）。
- 对象内的属性或方法之间使用逗号分隔。最后一个名值对后也可以有逗号，但在IE下会报错，所以尽量不要在最后一个属性或方法后加逗号。
- 属性名和值之间使用冒号分隔
- 如果将对象赋值给一个变量，不要忘了在右括号`}`之后补上分号

通过构造函数创建对象

JavaScript中没有类的概念，这给JavaScript带来了极大的灵活性，因为你不必提前知晓关于对象的任何信息，也不需要类的“蓝图”。但JavaScript同样具有构造函数，它的语法和Java或其他语言中基于类的对象创建非常类似。

你可以使用自定义的构造函数来创建实例对象，也可以使用内置构造函数来创建，比如Object()、Date()、String()等等。

下面这个例子展示了用两种等价的方法分别创建两个独立的实例对象：

```
// one way -- using a literal
var car = {goes: "far"};

// another way -- using a built-in constructor
// warning: this is an antipattern
var car = new Object();
car.goes = "far";
```

从这个例子中可以看到，直接量写法的一个明显优势是，它的代码更少。“创建对象的最佳模式是使用直接量”还有一个原因，它可以强调对象就是一个简单的可变的散列表，而不必一定派生自某个类。

另外一个使用直接量而不是Object构造函数创建实例对象的原因是，对象直接量不需要“作用域解析”（scope resolution）。因为新创建的实例有可能包含了一个本地的构造函数，当你调用Object()的时候，解析器需要顺着作用域链从当前作用域开始查找，直到找到全局Object构造函数为止。

获得对象的构造器

创建实例对象时能用对象直接量就不要使用new Object()构造函数，但有时你希望能继承别人写的代码，这时就需要了解构造函数的一个“特性”（也是不使用它的另一个原因），就是Object()构造函数可以接收参数，通过参数的设置可以把实例对象的创建委托给另一个内置构造函数，并返回另外一个实例对象，而这往往不是你所希望的。

下面的示例代码中展示了给new Object()传入不同的参数：数字、字符串和布尔值，最终得到的对象都是由不同的构造函数生成的：

```
// Warning: antipatterns ahead

// an empty object
var o = new Object();
console.log(o.constructor === Object); // true

// a number object
var o = new Object(1);
console.log(o.constructor === Number); // true
console.log(o.toFixed(2)); // "1.00"

// a string object
var o = new Object("I am a string");
console.log(o.constructor === String); // true
// normal objects don't have a substring()
// method but string objects do
console.log(typeof o.substring); // "function"

// a boolean object
var o = new Object(true);
console.log(o.constructor === Boolean); // true
```

Object()构造函数的这种特性会导致一些意想不到的结果，特别是当参数不确定的时候。最后再次提醒不要使用new Object()，尽可能的使用对象直接量来创建实例对象。

自定义构造函数

除了对象直接量和内置构造函数之外，你也可以通过自定义的构造函数来创建实例对象，正如下面的代码所示：

```
var adam = new Person("Adam");
adam.say(); // "I am Adam"
```

这里用了“类”Person创建了实例，这种写法看起来很像Java中的实例创建。两者的语法的确非常接近，但实际上JavaScript中没有类的概念，Person是一个函数。

Person构造函数是如何定义的呢？看下面的代码：

```
var Person = function (name) {
    this.name = name;
    this.say = function () {
        return "I am " + this.name;
    };
};
```

当你通过关键字new来调用这个构造函数时，函数体内将发生这些事情：

- 创建一个空对象，将它的引用赋给this，继承函数的原型。
- 通过this将属性和方法添加至这个对象
- 最后返回this指向的新对象（如果没有手动返回其他的对象）

用代码表示这个过程如下：

```
var Person = function (name) {
    // create a new object
    // using the object literal
    // var this = {};

    // add properties and methods
    this.name = name;
    this.say = function () {
        return "I am " + this.name;
    };

    //return this;
};
```

正如这段代码所示，say()方法添加至this中，结果是，不论何时调用new Person()，在内存中都会创建一个新函数（译注：所有Person的实例对象中的方法都是独占一块内存的）。显然这是效率很低的，因为所有实例的say()方法是一模一样的，因此没有必要“拷贝”多份。最好的办法是将方法添加至Person的原型中。

```
Person.prototype.say = function () {
    return "I am " + this.name;
};
```

我们将会在下一章里详细讨论原型和继承。现在只要记住将需要重用的成员和方法放在原型里即可。

关于构造函数的内部工作机制也会在后续章节中有更细致的讨论。这里我们只做概要的介绍。刚才提到，构造函数执行的时候，首先创建一个新对象，并将它的引用赋给this：

```
// var this = {};
```

事实并不完全是这样，因为“空”对象并不是真的空，这个对象继承了Person的原型，看起来更像：

```
// var this = Object.create(Person.prototype);
```

在后续章节会进一步讨论Object.create()。

构造函数的返回值

用new调用的构造函数总是会返回一个对象，默认返回this所指向的对象。如果构造函数内没有给this赋任何属性，则返回一个“空”对象（除了继承构造函数的原型之外，没有“自己的”属性）。

尽管我们不会在构造函数内写return语句，也会隐式返回this。但我们是可以返回任意指定的对象的，在下面的例子中就返回了新创建的that对象。

```
var Objectmaker = function () {

    // this `name` property will be ignored
    // because the constructor
    // decides to return another object instead
    this.name = "This is it";

    // creating and returning a new object
    var that = {};
    that.name = "And that's that";
    return that;
};

// test
var o = new Objectmaker();
console.log(o.name); // "And that's that"
```

我们看到，构造函数中其实是可以返回任意对象的，只要你返回的东西是对象即可。如果返回值不是对象（字符串、数字或布尔值），程序不会报错，但这个返回值被忽略，最终还是返回this所指的对象。

强制使用new的模式

我们知道，构造函数和普通的函数无异，只是通过new调用而已。那么如果调用构造函数时忘记new会发生什么呢？漏掉new不会产生语法错误也不会有运行时错误，但可能会造成逻辑错误，导致执行结果不符合预期。这是因为如果不写new的话，函数内的this会指向全局对象（在浏览器端this指向window）。

当构造函数内包含this.member之类的代码，并直接调用这个函数（省略new），实际会创建一个全局对象的属性member，可以通过window.member或member访问到它。这必然不是我们想要的结果，因为我们要努力确保全局命名空间的整洁干净。

```
// constructor
function Waffle() {
    this.tastes = "yummy";
}

// a new object
var good_morning = new Waffle();
console.log(typeof good_morning); // "object"
console.log(good_morning.tastes); // "yummy"

// antipattern:
// forgotten `new`
var good_morning = Waffle();
console.log(typeof good_morning); // "undefined"
console.log(window.tastes); // "yummy"
```

ECMAScript5中修正了这种非正常的行为逻辑。在严格模式中，this是不能指向全局对象的。如果在不支持ES5的JavaScript环境中，仍然后很多方法可以确保构造函数的行为即便在省略new调用时也不会出问题。

命名约定

最简单的选择是使用命名约定，前面的章节已经提到，构造函数名首字母大写（MyConstructor），普通函数和方法名首字母小写（myFunction）。

使用that

遵守命名约定的确能帮上一些忙，但约定毕竟不是强制，不能完全避免出错。这里给出了一种模式可以确保构造函数一定会按照构造函数的方式执行。不要将所有成员挂在this上，将它们挂在that上，并返回that。

```
function Waffle() {
var that = {};
    that.tastes = "yummy";
    return that;
}
```

如果要创建简单的实例对象，甚至不需要定义一个局部变量that，可以直接返回一个对象直接量，就像这样：

```
function Waffle() {
    return {
        tastes: "yummy"
    };
}
```

不管用什么方式调用它（使用new或直接调用），它同都会返回一个实例对象：

```
var first = new Waffle(),
    second = Waffle();
console.log(first.tastes); // "yummy"
console.log(second.tastes); // "yummy"
```

这种模式的问题是丢失了原型，因此在Waffle()的原型上的成员不会继承到这些实例对象中。

> 需要注意的是，这里用的that只是一种命名约定，that不是语言的保留字，可以将它替换为任何你喜欢的名字，比如self或me。

调用自身的构造函数

为了解决上述模式的问题，能够让实例对象继承原型属性，我们使用下面的方法。在构造函数中首先检查this是否是构造函数的实例，如果不是，再通过new调用构造函数，并将new的结果返回：

```
function Waffle() {

    if (!(this instanceof Waffle)) {
        return new Waffle();
    }
    this.tastes = "yummy";

}
Waffle.prototype.wantAnother = true;

// testing invocations
var first = new Waffle(),
    second = Waffle();

console.log(first.tastes); // "yummy"
console.log(second.tastes); // "yummy"

console.log(first.wantAnother); // true
console.log(second.wantAnother); // true
```

另一种检查实例的通用方法是使用arguments.callee，而不是直接将构造函数名写死在代码中：

```
if (!(this instanceof arguments.callee)) {
    return new arguments.callee();
}
```

这里需要说明的是，在任何函数内部都会自行创建一个arguments对象，它包含函数调用时传入的参数。同时arguments包含一个callee属性，指向它所在的正在被调用的函数。需要注意，ES5严格模式中是禁止使用arguments.callee的，因此最好对它的使用加以限制，并删除任何你能在代码中找到的实例（译注：这里作者的表述很委婉，其实作者更倾向于全面禁止使用arguments.callee）。

数组直接量

和其他的大多数一样，JavaScript中的数组也是对象。可以通过内置构造函数Array()来创建数组，类似对象直接量，数组也可以通过直接量形式创建。而且更推荐使用直接量创建数组。

这里的实例代码给出了创建两个具有相同元素的数组的两种方法，使用Array()和使用直接量模式：

```
// array of three elements
// warning: antipattern
var a = new Array("itsy", "bitsy", "spider");

// the exact same array
var a = ["itsy", "bitsy", "spider"];

console.log(typeof a); // "object", because arrays are objects
console.log(a.constructor === Array); // true
```

数组直接量语法

数组直接量写法非常简单：整个数组使用方括号括起来，数组元素之间使用逗号分隔。数组元素可以是任意类型，也包括数组和对象。

数组直接量语法简单直接、高雅美观。毕竟数组只是从位置0开始索引的值的集合，完全没必要包含构造器和new运算符的内容（代码会更多），保持简单即可。

有意思的数组构造器

我们对new Array()敬而远之原因是为了避免构造函数带来的陷阱。

如果给Array()构造器传入一个数字，这个数字并不会成为数组的第一个元素，而是设置数组的长度。也就是说，new Array(3)创建了一个长度为3的数组，而不是某个元素是3。如果你访问数组的任意元素都会得到undefined，因为元素并不存在。下面示例代码展示了直接量和构造函数的区别：

```
// an array of one element
var a = [3];
console.log(a.length); // 1
console.log(a[0]); // 3

// an array of three elements
var a = new Array(3);
console.log(a.length); // 3
console.log(typeof a[0]); // "undefined"
```

或许上面的情况看起来还不算是太严重的问题，但当 `new Array()` 的参数是一个浮点数而不是整数时则会导致严重的错误，这是因为数组的长度不可能是浮点数。

```
// using array literal
var a = [3.14];
console.log(a[0]); // 3.14

var a = new Array(3.14); // RangeError: invalid array length
console.log(typeof a); // "undefined"
```

为了避免在运行时动态创建数组时出现这种错误，强烈推荐使用数组直接量来代替new Array()。

> 有些人用Array()构造器来做一些有意思的事情，比如用来生成重复字符串。下面这行代码返字符串包含255个空格（请读者思考为什么不是256个空格）。`var white = new Array(256).join(' ');`

检查是不是数组

如果typeof的操作数是数组的话，将返回“object”。

```
console.log(typeof [1, 2]); // "object"
```

这个结果勉强说得过去，毕竟数组是一种对象，但对我们用处不大。往往你需要知道一个值是不是真正的数组。你可能见到过这种检查数组的方法：检查length属性、检查数组方法比如slice()等等。但这些方法非常脆弱，非数组的对象也可以拥有这些同名的属性。还有些人使用instanceof Array来判断数组，但这种方法在某些版本的IE里的多个iframe的场景中会出问题（译注：原因就是在不同iframe中创建的数组不会相互共享其prototype属性）。

ECMAScript 5定义了一个新的方法Array.isArray()，如果参数是数组的话就返回true。比如：

```
Array.isArray([]); // true

// trying to fool the check
// with an array-like object
Array.isArray({
    length: 1,
    "0": 1,
    slice: function () {}
}); // false
```

如果你的开发环境不支持ECMAScript5，可以通过Object.prototype.toString()方法来代替。如调用toString的call()方法并传入数组上下文，将返回字符串“[object Array]”。如果传入对象上下文，则返回字符串“[object Object]”。因此可以这样做：

```
if (typeof Array.isArray === "undefined") {
    Array.isArray = function (arg) {
        return Object.prototype.toString.call(arg) === "[object Array]";
    };
}
```

JSON

上文我们刚刚讨论过对象和数组直接量，你已经对此很熟悉了，现在我们将目光转向JSON。JSON（JavaScript Object Notation）是一种轻量级的数据交换格式。很多语言中都实现了JSON，特别是在JavaScript中。

JSON格式及其简单，它只是数组和对象直接量的混合写法，看一个JSON字符串的例子：

```
{"name": "value", "some": [1, 2, 3]}
```

JSON和对象直接量在语法上的唯一区别是，合法的JSON属性名均用引号包含。而在对象直接量中，只有属性名是非法的标识符时采用引号包含，比如，属性名中包含空格 `{"first name": "Dave"}` 。

在JSON字符串中，不能使用函数和正则表达式直接量。

使用JSON

在前面的章节中讲到，出于安全考虑，不推荐使用eval()来“粗糙的”解析JSON字符串。最好使用JSON.parse()方法，ES5中已经包含了这个方法，而且在现代浏览器的JavaScript引擎中已经内置支持JSON了。对于老旧的JavaScript引擎来说，你可以使用JSON.org所提供的JS文件（http://www.json.org/json2.js）来获得JSON对象和方法。

```
// an input JSON string
var jstr = '{"mykey": "my value"}';

// antipattern
var data = eval('(' + jstr + ')');

// preferred
var data = JSON.parse(jstr);

console.log(data.mykey); // "my value"
```

如果你已经在使用某个JavaScript库了，很可能库中提供了解析JSON的方法，就不必再额外引入JSON.org的库了，比如，如果你已经使用了YUI3，你可以这样：

```
// an input JSON string
var jstr = '{"mykey": "my value"}';

// parse the string and turn it into an object
// using a YUI instance
YUI().use('json-parse', function (Y) {
    var data = Y.JSON.parse(jstr);
    console.log(data.mykey); // "my value"
});
```

如果你使用的是jQuery，可以直接使用它提供的parseJSON()方法：

```
// an input JSON string
var jstr = '{"mykey": "my value"}';

var data = jQuery.parseJSON(jstr);
console.log(data.mykey); // "my value"
```

和JSON.parse()方法相对应的是JSON.stringify()。它将对象或数组（或任何原始值）转换为JSON字符串。

```
var dog = {
    name: "Fido",
    dob:new Date(),
    legs:[1,2,3,4]
};

var jsonstr = JSON.stringify(dog);

// jsonstr is now:
// {"name":"Fido","dob":"2010-04-11T22:36:22.436Z","legs":[1,2,3,4]}
```

正则表达式直接量

JavaScript中的正则表达式也是对象，可以通过两种方式创建它们：

- 使用new RegExp()构造函数
- 使用正则表达式直接量

下面的示例代码展示了创建正则表达式的两种方法，创建的正则用来匹配一个反斜杠（\）：

```
// regular expression literal
var re = /\\/gm;

// constructor
var re = new RegExp("\\\\", "gm");
```

显然正则表达式直接量写法的代码更短，且不必强制按照类构造器的思路来写。因此更推荐使用直接量写法。

另外，如果使用RegExp()构造函数写法，还需要考虑对引号和反斜杠进行转义，正如上段代码所示的那样，用了四个反斜杠来匹配一个反斜杠。这会增加正则表达式的长度，而且让正则变得难于理解和维护。刚开始学习正则表达式不是很容易，所以不要放弃任何一个简化它们的机会，所以要尽量使用直接量而不是通过构造函数来创建正则。

正则表达式直接量语法

正则表达式直接量使用两个斜线包裹起来，正则的主体部分不包括两端的斜线。在第二个斜线之后可以指定模式匹配的修饰符用以高级匹配，修饰符不需要引号引起来，JavaScript中有三个修饰符：

- g，全局匹配
- m，多行匹配
- i，忽略大小写的匹配

修饰符可以自由组合，而且顺序无关：

```
var re = /pattern/gmi;
```

使用正则表达式直接量可以让代码更加简洁高效，比如当调用String.prototype.prelace()方法时，可以传入正则表达式参数：

```
var no_letters = "abc123XYZ".replace(/[a-z]/gi, "");
console.log(no_letters); // 123
```

有一种不得不使用new RegExp()的情形，有时正则表达式是不确定的，直到运行时才能确定下来。

正则表达式直接量和Regexp()构造函数的另一个区别是，正则表达式直接量只在解析时创建一次正则表达式对象（译注：多次解析同一个正则表达式，会产生相同的实例对象）。如果在循环体内反复创建相同的正则表达式，则每个正则对象的所有属性（比如lastIndex）只会设置一次（译注：由于每次创建相同的实例对象，每个循环中的实例对象都是同一个，属性也自然相同），下面这个例子展示了两次都返回了相同的正则表达式的情形（译注：这里作者的表述只是针对ES3规范而言，下面这段代码在NodeJS、IE6-IE9、FireFox4、Chrome10、Safari5中运行结果和作者描述的不一致，Firefox 3.6中的运行结果和作者描述是一致的，原因可以在ECMAScript5规范第24页和第247页找到，也就是说在ECMAScript3规范中，用正则表达式创建的RegExp对象会共享同一个实例，而在ECMAScript5中则是两个独立的实例。而最新的Firefox4、Chrome和Safari5都遵循ECMAScript5标准，至于IE6-IE8都没有很好的遵循ECMAScript3标准，不过在这个问题上反而处理对了。很明显ECMAScript5的规范更符合开发者的期望）。

```
function getRE() {
    var re = /[a-z]/;
    re.foo = "bar";
    return re;
}

var reg = getRE(),
    re2 = getRE();

console.log(reg === re2); // true
reg.foo = "baz";
console.log(re2.foo); // "baz"
```

> 在ECMAScript5中这种情形有所改变，相同正则表达式直接量的每次计算都会创建新的实例对象，目前很多现代浏览器也对此做了纠正（译注：比如在Firefox4就纠正了Firefox3.6的这种“错误”）。

最后需要提一点，不带new调用RegExp()（作为普通的函数）和带new调用RegExp()是完全一样的。

原始值的包装对象

JavaScript中有五种原始类型：数字、字符串、布尔值、null和undefined。除了null和undefined之外，其他三种都有对应的“包装对象”（wrapper objects）。可以通过内置构造函数来生成包装对象，Number()、String()、和Boolean()。

为了说明数字原始值和数字对象之间的区别，看一下下面这个例子：

```
// a primitive number
var n = 100;
console.log(typeof n); // "number"

// a Number object
var nobj = new Number(100);
console.log(typeof nobj); // "object"
```

包装对象带有一些有用的属性和方法，比如，数字对象就带有toFixed()和toExponential()之类的方法。字符串对象带有substring()、chatAt()和toLowerCase()等方法以及length属性。这些方法非常方便，和原始值相比，这让包装对象具备了一定优势。其实原始值也可以调用这些方法，因为原始值会首先转换为一个临时对象，如果转换成功，则调用包装对象的方法。

```
// a primitive string be used as an object
var s = "hello";
console.log(s.toUpperCase()); // "HELLO"

// the value itself can act as an object
"monkey".slice(3, 6); // "key"

// same for numbers
(22 / 7).toPrecision(3); // "3.14"
```

因为原始值可以根据需要转换成对象，这样的话，也不必为了用包装对象的方法而将原始值手动“包装”成对象。比如，不必使用new String("hi")，直接使用"hi"即可。

```
// avoid these:
var s = new String("my string");
var n = new Number(101);
var b = new Boolean(true);

// better and simpler:
var s = "my string";
var n = 101;
var b = true;
```

不得不使用包装对象的一个原因是，有时我们需要对值进行扩充并保持值的状态。原始值毕竟不是对象，不能直接对其进行扩充（译注：比如 `1.property = 2` 会报错）。

```
// primitive string
var greet = "Hello there";

// primitive is converted to an object
// in order to use the split() method
greet.split(' ')[0]; // "Hello"

// attemting to augment a primitive is not an error
greet.smile = true;

// but it doesn't actually work
typeof greet.smile; // "undefined"
```

在这段示例代码中，greet只是临时转换成了对象，以保证访问其属性/方法时不会出错。另一方面，如果greet通过new String()定义为一个对象，那么扩充smile属性就会按照期望的那样执行。对字符串、数字或布尔值的扩充并不常见，除非你清楚自己想要什么，否则不必使用包装对象。

当省略new时，包装器将传给它的参数转换为原始值：

```
typeof Number(1); // "number"
typeof Number("1"); // "number"
typeof Number(new Number()); // "number"
typeof String(1); // "string"
typeof Boolean(1); // "boolean"
```

Error 对象

JavaScript中有很多内置的Error构造函数，比如Error()、SyntaxError()，TypeError()等等，这些“错误”通常和throw语句一起使用。这些构造函数创建的错误对象包含这些属性：

name

name属性是指创建这个对象的构造函数的名字，通常是“Errora”，有时会有特定的名字比如“RangeError”

message

创建这个对象时传入构造函数的字符串

错误对象还有其他一些属性，比如产生错误的行号和文件名，但这些属性是浏览器自行实现的，不同浏览器的实现也不一致，因此出于兼容性考虑，并不推荐使用这些属性。

另一方面，throw可以抛出任何对象，并不限于“错误对象”，因此你可以根据需要抛出自定义的对象。这些对象包含属性“name”和“message”或其他你希望传递给异常处理逻辑的信息，异常处理逻辑由catch语句指定。你可以灵活运用抛出的错误对象，将程序从错误状态恢复至正常状态。

```
try {
    // something bad happened, throw an error
    throw {
        name: "MyErrorType", // custom error type
        message: "oops",
        extra: "This was rather embarrassing",
        remedy: genericErrorHandler // who should handle it
    };
} catch (e) {
    // inform the user
    alert(e.message); // "oops"

    // gracefully handle the error
    e.remedy(); // calls genericErrorHandler()
}
```

通过new调用和省略new调用错误构造函数是一模一样的，他们都返回相同的错误对象。

小结

在本章里，我们讨论了多种直接量模式，它们是使用构造函数写法的替代方案，本章讲述了这些内容：

- 对象直接量写法——一种简洁优雅的定义对象的方法，名值对之间用逗号分隔，通过花括号包装起来
- 构造函数——内置构造函数（内置构造函数通常都有对应的直接量语法）和自定义构造函数。
- 一种强制函数以构造函数的模式执行（不管用不用new调用构造函数，都始终返回new出来的实例）的技巧
- 数组直接量写法——数组元素之间使用逗号分隔，通过方括号括起来
- JSON——是一种轻量级的数据交换格式
- 正则表达式直接量
- 避免使用其他的内置构造函数：String()、Number()、Boolean()以及不同种类的Error()构造器

通常除了Date()构造函数之外，其他的内置构造函数并不常用，下面的表格中对这些构造函数以及它们的直接量语法做了整理。

内置构造函数（不推荐）	直接量语法和原始值（推荐）
var o = new Object();	var o = {};
var a = new Array();	var a = [];
var re = new RegExp("[a-z]","g");	var re = /[a-z]/g;
var s = new String();	var s = "";
var n = new Number();	var n = 0;
var b = new Boolean();	var b = false;
throw new Error("uh-oh");	throw { name: "Error",message: "uh-oh"};或者throw Error("uh-oh");

函数

熟练运用函数是JavaScript程序员的必备技能，因为在JavaScript中函数实在是太常用了。它能够完成的任务种类非常之多，而在其他语言中则需要很多特殊的语法支持才能达到这种能力。

在本章将会介绍在JavaScript中定义函数的多种方式，包括函数表达式和函数声明、以及局部作用域和变量声明提前的工作原理。然后会介绍一些有用的模式，帮助你设计API（为你的函数提供更好的接口）、搭建代码架构（使用尽可能少的全局对象）、并优化性能（避免不必要的操作）。

现在让我们来一起揭秘JavaScript函数，我们首先从一些背景知识开始说起。

背景知识

JavaScript的函数具有两个主要特性，正是这两个特性让它们与众不同。第一个特性是，函数是一等对象（first-class object），第二个是函数提供作用域支持。

函数是对象，那么：

- 可以在程序执行时动态创建函数
- 可以将函数赋值给变量，可以将函数的引用拷贝至另一个变量，可以扩充函数，除了某些特殊场景外均可被删除。
- 可以将函数作为参数传入另一个函数，也可以被当作返回值返回。
- 函数可以包含自己的属性和方法

对于一个函数A来说，首先它是对象，拥有属性和方法，其中某个属性碰巧是另一个函数B，B可以接受函数作为参数，假设这个函数参数为C，当执行B的时候，返回另一个函数D。乍一看这里有一大堆相互关联的函数。当你开始习惯函数的许多用法时，你会惊叹原来函数是如此强大、灵活并富有表现力。通常说来，一说到JavaScript的函数，我们首先认为它是对象，它具有一个可以“执行”的特性，也就是说我们可以“调用”这个函数。

我们通过new Function()构造器来生成一个函数，这时可以明显看出函数是对象：

```
// antipattern
// for demo purposes only
var add = new Function('a, b', 'return a + b');
add(1, 2); // returns 3
```

在这段代码中，毫无疑问add()是一个对象，毕竟它是由构造函数创建的。这里并不推荐使用Function()构造器创建函数（和eval()一样糟糕），因为程序逻辑代码是以字符串的形式传入构造器的。这样的代码可读性差，写起来也很费劲，你不得不对逻辑代码中的引号做转义处

理，并需要特别关注为了让代码保持一定的可读性而保留的空格和缩进。

函数的第二个重要特性是它能提供作用域支持。在JavaScript中没有块级作用域（译注：在JavaScript1.7中提供了块级作用域部分特性的支持，可以通过let来声明块级作用域内的“局部变量”），也就是说不能通过花括号来创建作用域，JavaScript中只有函数作用域（译注：这里作者的表述只针对函数而言，此外JavaScript还有全局作用域）。在函数内所有通过var声明的变量都是局部变量，在函数外部是不可见的。刚才所指花括号无法提供作用域支持的意思是说，如果在if条件句内、或在for或while循环体内用var定义了变量，这个变量并不是属于if语句或for（while）循环的局部变量，而是属于它所在的函数。如果不在任何函数内部，它会成为全局变量。在第二章里提到我们要减少对全局命名空间的污染，那么使用函数则是控制变量的作用域的不二之选。

术语释义

首先我们先简单讨论下创建函数相关的术语，因为精确无歧义的术语约定和我们所讨论的各种模式一样重要。

看下这个代码片段：

```
// named function expression
var add = function add(a, b) {
    return a + b;
};
```

这段代码描述了一个函数，这种描述称为“带有命名的函数表达式”。

如果函数表达式将名字省略掉（比如下面的示例代码），这时它是“无名字的函数表达式”，通常我们称之为“匿名函数”，比如：

```
// function expression, a.k.a. anonymous function
var add = function (a, b) {
    return a + b;
};
```

因此“函数表达式”是一个更广义的概念，“带有命名的函数表达式”是函数表达式的一种特殊形式，仅仅当需要给函数定义一个可选的名字时使用。

当省略第二个add，它就成了无名字的函数表达式，这不会对函数定义和调用语法造成任何影响。带名字和不带名字唯一的区别是函数对象的name属性是否是一个空字符串。name属性属于语言的扩展（未在ECMA标准中定义），但很多环境都实现了。如果不省略第二个add，那么属性add.name则是"add"，name属性在用Firebug的调试过程中非常有用，还能让函数递归调用自身，其他情况可以省略它。

最后来看一下“函数声明”，函数声明的语法和其他语言中的语法非常类似：

```
function foo() {
    // function body goes here
}
```

从语法角度讲，带有命名的函数表达式和函数声明非常像，特别是当不需要将函数表达式赋值给一个变量的时候（在本章后面所讲到的回调模式中有类似的例子）。多数情况下，函数声明和带命名的函数表达式在外观上没有多少不同，只是它们在函数执行时对上下文的影响有所区别，下一小节会讲到。

两种语法的一个区别是末尾的分号。函数声明末尾不需要分号，而函数表达式末尾是需要分号的。推荐你始终不要丢掉函数表达式末尾的分号，即便JavaScript可以进行分号补全，也不要冒险这样做。

> 另外我们经常看到“函数直接量”。它用来表示函数表达式或带命名的函数表达式。由于这个术语是有歧义的，所以最好不要用它。

声明 vs 表达式：命名与提前

那么，到底应该用哪个呢？函数声明还是函数表达式？在不能使用函数声明语法的场景下，只能使用函数表达式了。下面这个例子中，我们给函数传入了另一个函数对象作为参数，以及给对象定义方法:

```
// this is a function expression,
// pased as an argument to the function `callMe`
callMe(function () {
    // I am an unnamed function expression
    // also known as an anonymous function
});

// this is a named function expression
callMe(function me() {
    // I am a named function expression
    // and my name is "me"
});

// another function expression
var myobject = {
    say: function () {
        // I am a function expression
    }
};
```

函数声明只能出现在“程序代码”中，也就是说在别的函数体内或在全局。这个定义不能赋值给变量或属性，同样不能作为函数调用的参数。下面这个例子是函数声明的合法用法，这里所有的函数foo()，bar()和local()都使用函数声明来定义：

```
// global scope
function foo() {}

function local() {
    // local scope
    function bar() {}
    return bar;
}
```

函数的name属性

选择函数定义模式的另一个考虑是只读属性name的可用性。尽管标准规范中并未规定，但很多运行环境都实现了name属性，在函数声明和带有名字的函数表达式中是有name的属性定义的。在匿名函数表达式中，则不一定有定义，这个是和实现相关的，在IE中是无定义的，在Firefox和Safari中是有定义的，但是值为空字符串。

```
function foo() {} // declaration
var bar = function () {}; // expression
var baz = function baz() {}; // named expression

foo.name; // "foo"
bar.name; // ""
baz.name; // "baz"
```

在Firebug或其他工具中调试程序时name属性非常有用，它可以用来显示当前正在执行的函数。同样可以通过name属性来递归的调用函数自身。如果你对这些场景不感兴趣，那么请尽可能的使用匿名函数表达式，这样会更简单、且冗余代码更少。

和函数声明相比而言，函数表达式的语法更能说明函数是一种对象，而不是某种特别的语言写法。

> 我们可以将一个带名字的函数表达式赋值给变量，变量名和函数名不同，这在技术上是可行的。比如：`var foo = function bar(){};`。然而，这种用法的行为在浏览器中的兼容性不佳（特别是IE中），因此并不推荐大家使用这种模式。

函数提前

通过前面的讲解，你可能以为函数声明和带名字的函数表达式是完全等价的。事实上不是这样，主要区别在于“声明提前”的行为。

> 术语“提前”并未在ECMAScript中定义，但是并没有其他更好的方法来描述这种行为了。

我们知道，不管在函数内何处声明变量，变量都会自动提前至函数体的顶部。对于函数来说亦是如此，因为他们也是一种对象，赋值给了变量。需要注意的是，函数声明定义的函数不仅能让声明提前，还能让定义提前，看一下这段示例代码：

```
// antipattern
// for illustration only

// global functions
function foo() {
    alert('global foo');
}
function bar() {
    alert('global bar');
}

function hoistMe() {

    console.log(typeof foo); // "function"
    console.log(typeof bar); // "undefined"

    foo(); // "local foo"
    bar(); // TypeError: bar is not a function

    // function declaration:
    // variable 'foo' and its implementation both get hoisted

    function foo() {
        alert('local foo');
    }

    // function expression:
    // only variable 'bar' gets hoisted
    // not the implementation
    var bar = function () {
        alert('local bar');
    };
}
hoistMe();
```

在这段代码中，和普通的变量一样，hoistMe()函数中的foo和bar被“搬运”到了顶部，覆盖了全局的foo和bar。不同之处在于，局部的foo()定义提前至顶部并能正常工作，尽管定义它的位置并不靠前。bar()的定义并未提前，只是声明提前了。因此当程序执行到bar()定义的位置之前，它的值都是undefined，并不是函数（防止当前上下文查找到作用域链上的全局的bar()，也就“覆盖”了全局的bar()）。

到目前为止我们介绍了必要的背景知识和函数定义相关的术语，下面开始介绍一些JavaScript所提供的函数相关的好的模式，我们从回调模式开始。同样，再次强调JavaScript函数的两个特殊特性，掌握这两点至关重要：

- 函数是对象
- 函数提供局部变量作用域

回调模式

函数是对象，也就意味着函数可以当作参数传入另外一个函数中。当你给函数writeCode()传入一个函数参数introduceBugs()，在某个时刻writeCode()执行了（或调用了）introduceBugs()。在这种情况下，我们说introduceBugs()是一个“回调函数”，简称“回调”：

```
function writeCode(callback) {
    // do something...
    callback();
    // ...
}

function introduceBugs() {
    // ... make bugs
}

writeCode(introduceBugs);
```

注意introduceBugs()是如何作为参数传入writeCode()的，当作参数的函数不带括号。括号的意思是执行函数，而这里我们希望传入一个引用，让writeCode()在合适的时机执行它（调用它）。

一个回调的例子

我们从一个例子开始，首先介绍无回调的情况，然后在作修改。假设你有一个通用的函数，用来完成某种复杂的逻辑并返回一大段数据。假设我们用findNodes()来命名这个通用函数，这个函数用来对DOM树进行遍历，并返回我所感兴趣的页面节点：

```
var findNodes = function () {
    var i = 100000, // big, heavy loop
        nodes = [], // stores the result
        found; // the next node found
    while (i) {
        i -= 1;
        // complex logic here...
        nodes.push(found);
    }
    return nodes;
};
```

保持这个函数的功能的通用性并一贯返回DOM节点组成的数组，并不会发生对节点的实际操作，这是一个不错的注意。可以将操作节点的逻辑放入另外一个函数中，比如放入一个hide()函数中，这个函数用来隐藏页面中的节点元素：

```
var hide = function (nodes) {
    var i = 0, max = nodes.length;
    for (; i < max; i += 1) {
        nodes[i].style.display = "none";
    }
};

// executing the functions
hide(findNodes());
```

这个实现的效率并不高，因为它将findNodes()所返回的节点数组重新遍历了一遍。最好在findNodes()中选择元素的时候就直接应用hide()操作，这样就能避免第二次的遍历，从而提高效率。但如果将hide()的逻辑写死在findNodes()的函数体内，findNodes()就变得不再通用了

（译注：如果我将hide()的逻辑替换成其他逻辑怎么办呢？），因为修改逻辑和遍历逻辑耦合在一起了。如果使用回调模式，则可以将隐藏节点的逻辑写入回调函数，将其传入findNodes()中适时执行：

```
// refactored findNodes() to accept a callback
var findNodes = function (callback) {
    var i = 100000,
        nodes = [],
        found;

    // check if callback is callable
    if (typeof callback !== "function") {
        callback = false;
    }
    while (i) {
        i -= 1;

        // complex logic here...

        // now callback:
        if (callback) {
            callback(found);
        }

        nodes.push(found);
    }
    return nodes;
};
```

这里的实现比较直接，findNodes()多作了一个额外工作，就是检查回调函数是否存在，如果存在的话就执行它。回调函数是可选的，因此修改后的findNodes()也是和之前一样使用，是可以兼容旧代码和旧API的。

这时hide()的实现就非常简单了，因为它不用对元素列表做任何遍历了：

```
// a callback function
var hide = function (node) {
    node.style.display = "none";
};

// find the nodes and hide them as you go
findNodes(hide);
```

正如代码中所示，回调函数可以是事先定义好的，也可以是一个匿名函数，你也可以将其称作main函数，比如这段代码，我们利用同样的通用函数findNodes()来完成显示元素的操作：

```
// passing an anonymous callback
findNodes(function (node) {
    node.style.display = "block";
});
```

回调和作用域

在上一个例子中，执行回调函数的写法是：

```
callback(parameters);
```

尽管这种写法可以适用大多数的情况，而且足够简单，但还有一些场景，回调函数不是匿名函数或者全局函数，而是对象的方法。如果回调函数中使用this指向它所属的对象，则回调逻辑往往并不像我们希望的那样执行。

假设回调函数是paint()，它是myapp的一个方法：

```
var myapp = {};
myapp.color = "green";
myapp.paint = function (node) {
    node.style.color = this.color;
};
```

函数findNodes()大致如下：

```
var findNodes = function (callback) {
    // ...
    if (typeof callback === "function") {
        callback(found);
    }
    // ...
};
```

当你调用findNodes(myapp.paint)，运行结果和我们期望的不一致，因为this.color未定义。因为findNodes()是全局函数，this指向的是全局对象。如果findNodes()是dom对象的方法（类似dom.findNodes()），那么回调函数内的this则指向dom，而不是myapp。

解决办法是，除了传入回调函数，还需将回调函数所属的对象当作参数传进去：

```
findNodes(myapp.paint, myapp);
```

同样需要修改findNodes()的逻辑，增加对传入的对象的绑定：

```
var findNodes = function (callback, callback_obj) {
    //...
    if (typeof callback === "function") {
        callback.call(callback_obj, found);
    }
    // ...
};
```

在后续的章节会对call()和apply()有更详细的讲述。

其实还有一种替代写法，就是将函数当作字符串传入findNodes()，这样就不必再写一次对象了，换句话说：

```
findNodes(myapp.paint, myapp);
```

可以写成：

```
findNodes("paint", myapp);
```

在findNodes()中的逻辑则需要修改为：

```
var findNodes = function (callback, callback_obj) {

    if (typeof callback === "string") {
        callback = callback_obj[callback];
    }

    //...
    if (typeof callback === "function") {
        callback.call(callback_obj, found);
    }
    // ...
};
```

异步事件监听

JavaScript中的回调模式已经是我们的家常便饭了，比如，如果你给网页中的元素绑定事件，则需要提供回调函数的引用，以便事件发生时能调用到它。这里有一个简单的例子，我们将console.log()作为回调函数绑定了document的点击事件：

```
document.addEventListener("click", console.log, false);
```

客户端浏览器中的大多数编程都是事件驱动的，当网页下载完成，则触发load事件，当用户和页面产生交互时也会触发多种事件，比如click、keypress、mouseover、mousemove等等。正是由于回调模式的灵活性，JavaScript天生适于事件驱动编程。回调模式能够让程序“异步”执行，换句话说，就是让程序不按顺序执行。

“不要打电话给我，我会打给你”，这是好莱坞很有名的一句话，很多电影都有这句台词。电影中的主角不可能同时应答很多个电话呼叫。在JavaScript的异步事件模型中也是同样的道理。电影中是留下电话号码，JavaScript中是提供一个回调函数，当时机成熟时就触发回调。有时甚至提供了很多回调，有些回调压根是没用的，但由于这个事件可能永远不会发生，因此这些回调的逻辑也不会执行。比如，假设你从此不再用“鼠标点击”，那么你之前绑定的鼠标点击的回调函数则永远也不会执行。

超时

另外一个最常用的回调模式是在调用超时函数时，超时函数是浏览器window对象的方法，共有两个：setTimeout()和setInterval()。这两个方法的参数都是回调函数。

```
var thePlotThickens = function () {
    console.log('500ms later...');
};
setTimeout(thePlotThickens, 500);
```

再次需要注意，函数thePlotThickens是作为变量传入setTimeout的，它不带括号，如果带括号的话则立即执行了，这里只是用到这个函数的引用，以便在setTimeout的逻辑中调用到它。也可以传入字符串“thePlotThickens()”，但这是一种反模式，和eval()一样不推荐使用。

库中的回调

回调模式非常简单，但又很强大。可以随手拈来灵活运用，因此这种模式在库的设计中也非常得宠。库的代码要尽可能的保持通用和重用，而回调模式则可帮助库的作者完成这个目标。你不必预料和实现你所想到的所有情形，因为这会让库变的膨胀而臃肿，而且大多数用户并不需要这些多余的特性支持。相反，你将精力放在核心功能的实现上，提供回调的入口作为“钩子”，可以让库的方法变得可扩展、可定制。

返回函数

函数是对象，因此当然可以作为返回值。也就是说，函数不一定非要返回一坨数据，函数可以返回另外一个定制好的函数，或者可以根据输入的不同按需创造另外一个函数。

这里有一个简单的例子：一个函数完成了某种功能，可能是一次性初始化，然后都基于这个返回值进行操作，这个返回值恰巧是另一个函数：

```
var setup = function () {
    alert(1);
    return function () {
        alert(2);
    };
};

// using the setup function
var my = setup(); // alerts 1
my(); // alerts 2
```

因为setup()把返回的函数作了包装，它创建了一个闭包，我们可以用这个闭包来存储一些私有数据，这些私有数据可以通过返回的函数进行操作，但在函数外部不能直接读取到这些私有数据。比如这个例子中提供了一个计数器，每次调用这个函数计数器都会加一：

```
var setup = function () {
    var count = 0;
    return function () {
        return (count += 1);
    };
};

// usage
var next = setup();
next(); // returns 1
next(); // 2
next(); // 3
```

自定义函数

我们动态定义函数，并将函数赋值给变量。如果将你定义的函数赋值给已经存在的函数变量的话，则新函数会覆盖旧函数。这样做的结果是，旧函数的引用就丢弃掉了，变量中所存储的引用值替换成了新的。这样看起来这个变量指代的函数逻辑就发生了变化，或者说函数进行了“重新定义”或“重写”。说起来有些拗口，实际上并不复杂，来看一个例子：

```
var scareMe = function () {
    alert("Boo!");
    scareMe = function () {
        alert("Double boo!");
    };
};
// using the self-defining function
scareMe(); // Boo!
scareMe(); // Double boo!
```

当函数中包含一些初始化操作，并希望这些初始化只执行一次，那么这种模式是非常适合这个场景的。因为能避免的重复执行则尽量避免，函数的一部分可能再也不会执行到。在这个场景中，函数执行一次后就被重写为另外一个函数了。

使用这种模式可以帮助提高应用的执行效率，因为重新定义的函数执行更少的代码。

> 这种模式的另外一个名字是“函数的懒惰定义”，因为直到函数执行一次后才重新定义，可以说它是“某个时间点之后才存在”，简称“懒惰定义”。

这种模式有一种明显的缺陷，就是之前给原函数添加的功能在重定义之后都丢失了。如果将这个函数定义为不同的名字，函数赋值给了很多不同的变量，或作为对象的方法使用，那么新定义的函数有可能不会执行，原始的函数会照旧执行（译注：由于函数的赋值是引用的赋值，函数赋值给多个变量只是将引用赋值给了多个变量，当某一个变量定义了新的函数，也只是变量的引用值发生变化，原函数本身依旧存在，当程序中存在某个变量的引用还是旧函数的话，旧函数还是会依旧执行）。

让我们来看一个例子，scareMe()函数在这里作为一等对象来使用：

1. 给他增加了一个属性
2. 函数对象赋值给一个新变量

3. 函数依旧可以作为方法来调用

看一下这段代码：

```
// 1. adding a new property
scareMe.property = "properly";

// 2. assigning to a different name
var prank = scareMe;

// 3. using as a method
var spooky = {
    boo: scareMe
};

// calling with a new name
prank(); // "Boo!"
prank(); // "Boo!"
console.log(prank.property); // "properly"

// calling as a method
spooky.boo(); // "Boo!"
spooky.boo(); // "Boo!"
console.log(spooky.boo.property);    // "properly"

// using the self-defined function
scareMe(); // Double boo!
scareMe(); // Double boo!
console.log(scareMe.property); // undefined
```

从结果来看，当自定义函数被赋值给一个新的变量的时候，这段使用自定义函数的代码的执行结果与我们期望的结果可能并不一样。每当prank()运行的时候，它都弹出“Boo!”。同时它也重写了scareMe()函数，但是prank()自己仍然能够使用之前的定义，包括属性property。在这个函数被作为spooky对象的boo()方法调用的时候，结果也一样。所有的这些调用，在第一次的时候就已经修改了全局的scareMe()的指向，所以当它最终被调用的时候，它的函数体已经被修改为弹出“Double boo”。它也就不能获取到新添加的属性“property”。

立即执行的函数

立即执行的函数是一种语法模式，它会使函数在定义后立即执行。看这个例子：

```
(function () {
    alert('watch out!');
}());
```

这种模式本质上只是一个在创建后就被执行的函数表达式（具名或者匿名）。“立即执行的函数”这种说法并没有在ECMAScript标准中被定义，但它作为一个名词，有助于我们的描述和讨论。

这种模式由以下几个部分组成：

- 使用函数表达式定义一个函数。（不能使用函数声明。）
- 在最后加入一对括号，这会使函数立即被执行。

- 把整个函数包裹到一对括号中（只在没有将函数赋值给变量时需要）。

下面这种语法也很常见（注意右括号的位置），但是JSLint倾向于第一种：

```
(function () {
    alert('watch out!');
})();
```

这种模式很有用，它为我们提供一个作用域的沙箱，可以在执行一些初始化代码的时候使用。设想这样的场景：当页面加载的时候，你需要运行一些代码，比如绑定事件、创建对象等等。所有的这些代码都只需要运行一次，所以没有必要创建一个带有名字的函数。但是这些代码需要一些临时变量，而这些变量在初始化完之后又不会再次用到。显然，把这些变量作为全局变量声明是不合适的。正因为如此，我们才需要立即执行的函数。它可以把你所有的代码包裹到一个作用域里面，而不会暴露任何变量到全局作用域中：

```
(function () {

    var days = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'],
        today = new Date(),
        msg = 'Today is ' + days[today.getDay()] + ', ' + today.getDate();

    alert(msg);

}()); // "Today is Fri, 13"
```

如果这段代码没有被包裹到立即执行函数中，那么变量days、today、msg都会是全局变量，而这些变量仅仅是由因为初始化而遗留下来的垃圾，没有任何用处。

立即执行的函数的参数

立即执行的函数也可以接受参数，看这个例子：

```
// prints:
// I met Joe Black on Fri Aug 13 2010 23:26:59 GMT-0800 (PST)

(function (who, when) {

    console.log("I met " + who + " on " + when);

}("Joe Black", new Date()));
```

通常的做法，会把全局对象当作一个参数传给立即执行的函数，以保证在函数内部也可以访问到全局对象，而不是使用window对象，这样可以使得代码在非浏览器环境中使用时更具可移植性。

值得注意的是，一般情况下尽量不要给立即执行的函数传入太多的参数，否则会有一件麻烦的事情，就是你在阅读代码的时候需要频繁地上下滚动代码。

立即执行的函数的返回值

和其它的函数一样，立即执行的函数也可以返回值，并且这些返回值也可以被赋值给变量：

```
var result = (function () {
    return 2 + 2;
}());
```

如果省略括号的话也可以达到同样的目的，因为如果需要将返回值赋给变量，那么第一对括号就不是必需的。省略括号的代码是这样子：

```
var result = function () {
    return 2 + 2;
}();
```

这种写法更简洁，但是同时也容易造成误解。如果有人在阅读代码的时候忽略了最后的一对括号，那么他会以为result指向了一个函数。而事实上result是指向这个函数运行后的返回值，在这个例子中是4。

还有一种写法也可以得到同样的结果：

```
var result = (function () {
    return 2 + 2;
})();
```

前面的例子中，立即执行的函数返回的是一个基本类型的数值。但事实上，除了基本类型以外，一个立即执行的函数可以返回任意类型的值，甚至返回一个函数都可以。你可以利用立即执行的函数的作用域来存储一些私有的数据，这些数据只能在返回的内层函数中被访问。

在下面的例子中，立即执行的函数的返回值是一个函数，这个函数会简单地返回res的值，并且它被赋给了变量getResult。而res是一个预先计算好的变量，它被存储在立即执行函数的闭包中：

```
var getResult = (function () {
    var res = 2 + 2;
    return function () {
        return res;
    };
}());
```

在定义一个对象的属性的时候也可以使用立即执行的函数。设想一下这样的场景：你需要定义一个对象的属性，这个属性在对象的生命周期中都不会改变，但是在定义之前，你需要做一点额外的工作来得到正确的值。这种情况下你就可以使用立即执行的函数来包裹那些额外的工作，然后将它的返回值作为对象属性的值。下面是一个例子：

```
var o = {
    message: (function () {
        var who = "me",
            what = "call";
        return what + " " + who;
    }()),
    getMsg: function () {
        return this.message;
    }
};

// usage
o.getMsg(); // "call me"
o.message; // "call me"
```

在这个例子中，o.message是一个字符串，而不是一个函数，但是它需要一个函数在脚本载入后来得到这个属性值。

好处和用法

立即执行的函数应用很广泛。它可以帮助我们做一些不想留下全局变量的工作。所有定义的变量都只是立即执行的函数的本地变量，你完全不用担心临时变量会污染全局对象。

> 立即执行的函数还有一些名字，比如“自调用函数”或者“自执行函数”，因为这些函数会在被定义后立即执行自己。

这种模式也经常被用到书签代码中，因为书签代码会在任何一个页面运行，所以需要非常苛刻地保持全局命名空间干净。

这种模式也可以让你包裹一些独立的特性到一个封闭的模块中。设想你的页面是静态的，在没有JavaScript的时候工作正常，然后，本着渐进增强的精神，你给页面加入了一点增加代码。这时候，你就可以把你的代码（也可以叫“模块”或者“特性”）放到一个立即执行的函数中并且保证页面在有没有它的时候都可以正常工作。然后你就可以加入更多的增强特性，或者对它们进行移除、进行独立测试或者允许用户禁用等等。

你可以使用下面的模板定义一段函数代码，我们叫它module1：

```
// module1 defined in module1.js
(function () {

    // all the module 1 code ...

}());
```

套用这个模板，你就可以编写其它的模块。然后在发布到线上的时候，你就可以决定在这个时间节点上哪些特性是可以使用的，然后使用发布脚本将它们打包上线。

立即初始化的对象

还有另外一种可以避免污染全局作用域的方法，和前面描述的立即执行的函数相似，叫做“立即初始化的对象”模式。这种模式使用一个带有init()方法的对象来实现，这个方法在对象被创建后立即执行。初始化的工作由init()函数来完成。

下面是一个立即初始化的对象模式的例子：

```
({
    // here you can define setting values
    // a.k.a. configuration constants
    maxwidth: 600,
    maxheight: 400,

    // you can also define utility methods
    gimmeMax: function () {
        return this.maxwidth + "x" + this.maxheight;
    },

    // initialize
    init: function () {
        console.log(this.gimmeMax());
        // more init tasks...
    }
}).init();
```

在语法上，当你使用这种模式的时候就像在使用对象字面量创建一个普通对象一样。除此之外，还需要将对象字面量用括号括起来，这样能让JavaScript引擎知道这是一个对象字面量，而不是一个代码块（if或者for循环之类）。在括号后面，紧接着就执行了init()方法。

你也可以将对象字面量和init()调用一起写到括号里面。简单地说，下面两种语法都是有效的：

```
({...}).init();
({...}.init());
```

这种模式的好处和自动执行的函数模式是一样的：在做一些一次性的初始化工作的时候保护全局作用域不被污染。从语法上看，这种模式似乎比只包含一段代码在一个匿名函数中要复杂一些，但是如果你的初始化工作比较复杂（这种情况很常见），它会给整个初始化工作一个比较清晰的结构。比如，一些私有的辅助性函数可以被很轻易地看出来，因为它们是这个临时对象的属性，但是如果是在立即执行的函数模式中，它们很可能只是一些散落的函数。

这种模式的一个弊端是，JavaScript压缩工具可能不能像压缩一段包裹在函数中的代码一样有效地压缩这种模式的代码。这些私有的属性和方法不被会重命名为一些更短的名字，因为从压缩工具的角度来看，保证压缩的可靠性更重要。在写作本书的时候，Google出品的Closure Compiler的“advanced”模式是唯一会重命名立即初始化的对象的属性的压缩工具。一个压缩后的样例是这样：

```
({d:600,c:400,a:function(){return this.d+"x"+this.c},b:function(){console.log(this.a())}}
```

> 这种模式主要用于一些一次性的工作，并且在init()方法执行完后就无法再次访问到这个对象。如果希望在这些工作完成后保持对对象的引用，只需要简单地在init()的末尾加上return this;即可。

条件初始化

条件初始化（也叫条件加载）是一种优化模式。当你知道某种条件在整个程序生命周期中都不会变化的时候，那么对这个条件的探测只做一次就很有意义。浏览器探测（或者特征检测）是一个典型的例子。

举例说明，当你探测到XMLHttpRequest被作为一个本地对象支持时，就知道浏览器不会在程序执行过程中改变这一情况，也不会出现突然需要去处理ActiveX对象的情况。当环境不发生变化的时候，你的代码就没有必要在需要在每次XHR对象时探测一遍（并且得到同样的结果）。

另外一些可以从条件初始化中获益的场景是获得一个DOM元素的computed styles或者是绑定事件处理函数。大部分程序员在他们的客户端编程生涯中都编写过事件绑定和取消绑定相关的组件，像下面的例子：

```
// BEFORE
var utils = {
    addListener: function (el, type, fn) {
        if (typeof window.addEventListener === 'function') {
            el.addEventListener(type, fn, false);
        } else if (typeof document.attachEvent === 'function') { // IE
            el.attachEvent('on' + type, fn);
        } else { // older browsers
            el['on' + type] = fn;
        }
    },
    removeListener: function (el, type, fn) {
        // pretty much the same...
    }
};
```

这段代码的问题就是效率不高。每当你执行utils.addListener()或者utils.removeListener()时，同样的检查都会被重复执行。

如果使用条件初始化，那么浏览器探测的工作只需要在初始化代码的时候执行一次。在初始化的时候，代码探测一次环境，然后重新定义这个函数在剩下来的程序生命周期中应该怎样工作。下面是一个例子，看看如何达到这个目的：

```
// AFTER

// the interface
var utils = {
    addListener: null,
    removeListener: null
};

// the implementation
if (typeof window.addEventListener === 'function') {
    utils.addListener = function (el, type, fn) {
        el.addEventListener(type, fn, false);
    };
    utils.removeListener = function (el, type, fn) {
        el.removeEventListener(type, fn, false);
    };
} else if (typeof document.attachEvent === 'function') { // IE
    utils.addListener = function (el, type, fn) {
        el.attachEvent('on' + type, fn);
    };
    utils.removeListener = function (el, type, fn) {
        el.detachEvent('on' + type, fn);
    };
} else { // older browsers
    utils.addListener = function (el, type, fn) {
        el['on' + type] = fn;
    };
    utils.removeListener = function (el, type, fn) {
        el['on' + type] = null;
    };
}
```

说到这里，要特别提醒一下关于浏览器探测的事情。当你使用这个模式的时候，不要对浏览器特性过度假设。举个例子，如果你探测到浏览器不支持window.addEventListener，不要假设这个浏览器是IE，也不要认为它不支持原生的XMLHttpRequest，虽然这个结论在整个浏览器历史上的某个点是正确的。当然，也有一些情况是可以放心地做一些特性假设的，比如.addEventListener和.removeEventListerner，但是通常来讲，浏览器的特性在发生变化时都是独立的。最好的策略就是分别探测每个特性，然后使用条件初始化，使这种探测只做一次。

函数属性——Memoization模式

函数也是对象，所以它们可以有属性。事实上，函数也确实本来就有一些属性。比如，对一个函数来说，不管是用什么语法创建的，它会自动拥有一个length属性来标识这个函数期待接受的参数个数：

```
function func(a, b, c) {}
console.log(func.length); // 3
```

任何时候都可以给函数添加自定义属性。添加自定义属性的一个有用场景是缓存函数的执行结果（返回值），这样下次同样的函数被调用的时候就不需要再做一次那些可能很复杂的计算。缓存一个函数的运行结果也就是为大家所熟知的Memoization。

在下面的例子中，myFunc函数创建了一个cache属性，可以通过myFunc.cache访问到。这个cache属性是一个对象（hash表），传给函数的参数会作为对象的key，函数执行结果会作为对象的值。函数的执行结果可以是任何的复杂数据结构：

```
var myFunc = function (param) {
    if (!myFunc.cache[param]) {
        var result = {};
        // ... expensive operation ...
        myFunc.cache[param] = result;
    }
    return myFunc.cache[param];
};

// cache storage
myFunc.cache = {};
```

上面的代码假设函数只接受一个参数param，并且这个参数是基本类型（比如字符串）。如果你有更多更复杂的参数，则通常需要对它们进行序列化。比如，你需要将arguments对象序列化为JSON字符串，然后使用JSON字符串作为cache对象的key：

```
var myFunc = function () {

    var cachekey = JSON.stringify(Array.prototype.slice.call(arguments)),
        result;

    if (!myFunc.cache[cachekey]) {
        result = {};
        // ... expensive operation ...
        myFunc.cache[cachekey] = result;
    }
    return myFunc.cache[cachekey];
};

// cache storage
myFunc.cache = {};
```

需要注意的是，在序列化的过程中，对象的“标识”将会丢失。如果你有两个不同的对象，却碰巧有相同的属性，那么他们会共享同样的缓存内容。

前面代码中的函数名还可以使用arguments.callee来替代，这样就不用将函数名硬编码。不过尽管现阶段这个办法可行，但是仍然需要注意，arguments.callee在ECMAScript 5的严格模式中是不被允许的：

```
var myFunc = function (param) {

    var f = arguments.callee,
        result;

    if (!f.cache[param]) {
        result = {};
        // ... expensive operation ...
        f.cache[param] = result;
    }
    return f.cache[param];
};

// cache storage
myFunc.cache = {};
```

配置对象

配置对象模式是一种提供更简洁的API的方法，尤其是当你正在写一个即将被其它程序调用的类库之类的代码的时候。

软件在开发和维护过程中需要不断改变是一个不争的事实。这样的事情总是以一些有限的需求开始，但是随着开发的进行，越来越多的功能会不断被加进来。

设想一下你正在写一个名为addPerson()的函数，它接受一个姓和一个名，然后在列表中加入一个人：

```
function addPerson(first, last) {...}
```

然后你意识到，生日也必须要存储，此外，性别和地址也作为可选项存储。所以你修改了函数，添加了一些新的参数（还得非常小心地将可选参数放到最后）：

```
function addPerson(first, last, dob, gender, address) {...}
```

这个时候，函数已经显得有点长了。然后，你又被告知需要添加一个用户名，并且不是可选的。现在这个函数的调用者需要将所有的可选参数传进来，并且得非常小心地保证不弄混参数的顺序：

```
addPerson("Bruce", "Wayne", new Date(), null, null, "batman");
```

传一大串的参数真的很不方便。一个更好的办法就是将它们替换成一个参数，并且把这个参数弄成对象；我们叫它conf，是“configuration”（配置）的缩写：

```
addPerson(conf);
```

然后这个函数的使用者就可以这样：

```
var conf = {
    username: "batman",
    first: "Bruce",
    last: "Wayne"
};
addPerson(conf);
```

配置对象模式的好处是：

- 不需要记住参数的顺序
- 可以很安全地跳过可选参数
- 拥有更好的可读性和可维护性
- 更容易添加和移除参数

配置对象模式的坏处是：

- 需要记住参数的名字
- 参数名字不能被压缩

举些实例，这个模式对创建DOM元素的函数或者是给元素设定CSS样式的函数会非常实用，因为元素和CSS样式可能会有很多但是大部分可选的属性。

柯里化 （Curry）

在本章剩下的部分，我们将讨论一下关于柯里化和部分应用的话题。但是在我们开始这个话题之前，先看一下到底什么是函数应用。

函数应用

在一些纯粹的函数式编程语言中，对函数的描述不是被调用（called或者invoked），而是被应用（applied）。在JavaScript中也有同样的东西——我们可以使用Function.prototype.apply()来应用一个函数，因为在JavaScript中，函数实际上是对象，并且他们拥有方法。

下面是一个函数应用的例子：

```
// define a function
var sayHi = function (who) {
    return "Hello" + (who ? ", " + who : "") + "!";
};

// invoke a function
sayHi(); // "Hello"
sayHi('world'); // "Hello, world!"

// apply a function
sayHi.apply(null, ["hello"]); // "Hello, hello!"
```

从上面的例子中可以看出来，调用一个函数和应用一个函数有相同的结果。apply()接受两个参数：第一个是在函数内部绑定到this上的对象，第二个是一个参数数组，参数数组会在函数内部变成一个类似数组的arguments对象。如果第一个参数为null，那么this将指向全局对象，这正是当你调用一个函数（且这个函数不是某个对象的方法）时发生的事情。

当一个函数是一个对象的方法时，我们不再像前面的例子一样传入null。（译注：主要是为了保证方法中的this绑定到一个有效的对象而不是全局对象。）在下面的例子中，对象被作为第一个参数传给apply()：

```
var alien = {
    sayHi: function (who) {
        return "Hello" + (who ? ", " + who : "") + "!";
    }
};

alien.sayHi('world'); // "Hello, world!"
sayHi.apply(alien, ["humans"]); // "Hello, humans!"
```

在这个例子中，sayHi()中的this指向alien。而在上一个例子中，this是指向的全局对象。（译注：这个例子的代码有误，最后一行的sayHi并不能访问到alien的sayHi方法，需要使用alien.sayHi.apply(alien, ["humans"])才可正确运行。另外，在sayHi中也没有出现this。）

正如上面两个例子所展现出来的一样，我们将所谓的函数调用当作函数应用的一种语法糖并没有什么太大的问题。

需要注意的是，除了apply()之外，Function.prototype对象还有一个call()方法，但是它仍然只是apply()的一种语法糖。（译注：这两个方法的区别在于，apply()只接受两个参数，第二个参数为需要传给函数的参数数组，而call()则接受任意多个参数，从第二个开始将参数依次传给函数。）不过有种情况下使用这个语法糖会更好：当你的函数只接受一个参数的时候，你可以省去为唯一的一个元素创建数组的工作：

```
// the second is more efficient, saves an array
sayHi.apply(alien, ["humans"]); // "Hello, humans!"
sayHi.call(alien, "humans"); // "Hello, humans!"
```

部分应用

现在我们知道了，调用一个函数实际上就是给它应用一堆参数，那是否能够只传一部分参数而不传全部呢？这实际上跟我们手工处理数学函数非常类似。

假设已经有了一个add()函数，它的工作是把x和y两个数加到一起。下面的代码片段展示了当x为5、y为4时的计算步骤：

```
// for illustration purposes
// not valid JavaScript

// we have this function
function add(x, y) {
    return x + y;
}

// and we know the arguments
add(5, 4);

// step 1 -- substitute one argument
function add(5, y) {
    return 5 + y;
}

// step 2 -- substitute the other argument
function add(5, 4) {
    return 5 + 4;
}
```

在这个代码片段中，step 1和step 2并不是有效的JavaScript代码，但是它展示了我们手工计算的过程。首先获得第一个参数的值，然后将未知的x和已知的值5替换到函数中。然后重复这个过程，直到替换掉所有的参数。

step 1是一个所谓的部分应用的例子：我们只应用了第一个参数。当你执行一个部分应用的时候并不能获得结果（或者是解决方案），取而代之的是另一个函数。

下面的代码片段展示了一个虚拟的partialApply()方法的用法：

```
var add = function (x, y) {
    return x + y;
};

// full application
add.apply(null, [5, 4]); // 9

// partial application
var newadd = add.partialApply(null, [5]);
// applying an argument to the new function
newadd.apply(null, [4]); // 9
```

正如你所看到的一样，部分应用给了我们另一个函数，这个函数可以在稍后调用的时候接受其它的参数。这实际上跟add(5)(4)是等价的，因为add(5)返回了一个函数，这个函数可以使用(4)来调用。我们又一次看到，熟悉的add(5, 4)也差不多是add(5)(4)的一种语法糖。

现在，让我们回到地球：并不存在这样的一个partialApply()函数，并且函数的默认表现也不会像上面的例子中那样。但是你完全可以自己去写，因为JavaScript的动态特性完全可以做到这样。

让函数理解并且处理部分应用的过程，叫柯里化（Currying）。

柯里化（Currying）

柯里化和辛辣的印度菜可没什么关系；它来自数学家Haskell Curry。（Haskell编程语言也是因他而得名。）柯里化是一个变换函数的过程。柯里化的另外一个名字也叫schönfinkelisation，来自另一位数学家——Moses Schönfinkelisation——这种变换的最初发明者。

所以我们怎样对一个函数进行柯里化呢？其它的函数式编程语言也许已经原生提供了支持并且所有的函数已经默认柯里化了。在JavaScript中我们可以修改一下add()函数使它柯里化，然后支持部分应用。

来看一个例子：

```
// a curried add()
// accepts partial list of arguments
function add(x, y) {
    var oldx = x, oldy = y;
    if (typeof oldy === "undefined") { // partial
        return function (newy) {
            return oldx + newy;
        };
    }
    // full application
    return x + y;
}

// test
typeof add(5); // "function"
add(3)(4); // 7

// create and store a new function
var add2000 = add(2000);
add2000(10); // 2010
```

在这段代码中，第一次调用add()时，在返回的内层函数那里创建了一个闭包。这个闭包将原来的x和y的值存储到了oldx和oldy中。当内层函数执行的时候，oldx会被使用。如果没有部分应用，即x和y都传了值，那么这个函数会简单地将他们相加。这个add()函数的实现跟实际情况比起来有些冗余，仅仅是为了更好地说明问题。下面的代码片段中展示了一个更简洁的版本，没有oldx和oldy，因为原始的x已经被存储到了闭包中，此外我们复用了y作为本地变量，而不用像之前那样新定义一个变量newy：

```
// a curried add
// accepts partial list of arguments
function add(x, y) {
    if (typeof y === "undefined") { // partial
        return function (y) {
            return x + y;
        };
    }
    // full application
    return x + y;
}
```

在这些例子中，add()函数自己处理了部分应用。有没有可能用一种更为通用的方式来做同样的事情呢？换句话说，我们能不能对任意一个函数进行处理，得到一个新函数，使它可以处理部分参数？下面的代码片段展示了一个通用函数的例子，我们叫它schonfinkelize()，正是

用来做这个的。我们使用schonfinkelize()这个名字，一部分原因是它比较难发音，另一部分原因是它听起来比较像动词（使用“curry”则不是那么明确），而我们刚好需要一个动词来表明这是一个函数转换的过程。

这是一个通用的柯里化函数：

```
function schonfinkelize(fn) {
    var slice = Array.prototype.slice,
    stored_args = slice.call(arguments, 1);
    return function () {
        var new_args = slice.call(arguments),
        args = stored_args.concat(new_args);
        return fn.apply(null, args);
    };
}
```

这个schonfinkelize可能显得比较复杂了，只是因为在JavaScript中arguments不是一个真的数组。从Array.prototype中借用slice()方法帮助我们将arguments转换成数组，以便能更好地对它进行操作。当schonfinkelize()第一次被调用的时候，它使用slice变量存储了对slice()方法的引用，同时也存储了调用时的除去第一个之外的参数（stored_args），因为第一个参数是要被柯里化的函数。schonfinkelize()返回了一个函数。当这个返回的函数被调用的时候，它可以（通过闭包）访问到已经存储的参数stored_args和slice。新的函数只需要合并老的部分应用的参数（stored_args）和新的参数（new_args），然后将它们应用到原来的函数fn（也可以在闭包中访问到）即可。

现在有了通用的柯里化函数，就可以做一些测试了：

```
// a normal function
function add(x, y) {
    return x + y;
}

// curry a function to get a new function
var newadd = schonfinkelize(add, 5);
newadd(4); // 9

// another option -- call the new function directly
schonfinkelize(add, 6)(7); // 13
```

用来做函数转换的schonfinkelize()并不局限于单个参数或者单步的柯里化。这里有些更多用法的例子：

```
// a normal function
function add(a, b, c, d, e) {
    return a + b + c + d + e;
}

// works with any number of arguments
schonfinkelize(add, 1, 2, 3)(5, 5); // 16

// two-step currying
var addOne = schonfinkelize(add, 1);
addOne(10, 10, 10, 10); // 41
var addSix = schonfinkelize(addOne, 2, 3);
addSix(5, 5); // 16
```

什么时候使用柯里化

当你发现自己在调用同样的函数并且传入的参数大部分都相同的时候，就是考虑柯里化的理想场景了。你可以通过传入一部分的参数动态地创建一个新的函数。这个新函数会存储那些重复的参数（所以你不需要再每次都传入），然后再在调用原始函数的时候将整个参数列表补全，正如原始函数期待的那样。

小结

在JavaScript中，开发者对函数的理解和运用的要求是比较苛刻的。在本章中，主要讨论了有关函数的一些背景知识和术语。介绍了JavaScript函数中两个重要的特性，也就是：

1. 函数是一等对象，他们可以被作为值传递，也可以拥有属性和方法。
2. 函数拥有本地作用域，而大括号不产生块级作用域。另外需要注意的是，变量的声明会被提前到本地作用域顶部。

创建一个函数的语法有：

1. 带有名字的函数表达式
2. 函数表达式（和上一种一样，但是没有名字），也就是为大家熟知的“匿名函数”
3. 函数声明，与其它语言的函数语法相似

在介绍完背景和函数的语法后，介绍了一些有用的模式，按分类列出：

1. API模式，它们帮助我们为函数给出更干净的接口，包括：
    - 回调模式

        ```
        传入一个函数作为参数
        ```

    - 配置对象

        ```
         帮助保持函数的参数数量可控
        ```

- 返回函数

```
函数的返回值是另一个函数
```

- 柯里化

```
新函数在已有函数的基础上再加上一部分参数构成
```

2. 初始化模式，这些模式帮助我们用一种干净的、结构化的方法来做一些初始化工作（在web页面和应用中非常常见），通过一些临时变量来保证不污染全局命名空间。这些模式包括：

   - 立即执行的函数

   ```
   当它们被定义后立即执行
   ```

   - 立即初始化的对象

   ```
   初始化工作被放入一个匿名对象，这个对象提供一个可以立即被执行的方法
   ```

   - 条件初始化

   ```
   使分支代码只在初始化的时候执行一次，而不是在整个程序生命周期中反复执行
   ```

3. 性能模式，这些模式帮助提高代码的执行速度，包括：

   - Memoization

   ```
   利用函数的属性，使已经计算过的值不用再次计算
   ```

   - 自定义函数

   ```
   重写自身的函数体，使第二次及后续的调用做更少的工作
   ```

对象创建模式

在JavaScript中创建对象很容易——可以通过使用对象直接量或者构造函数。本章将在此基础上介绍一些常用的对象创建模式。

JavaScript语言本身简单、直观，通常也没有其他语言那样的语法特性：命名空间、模块、包、私有属性以及静态成员。本章将介绍一些常用的模式，以此实现这些语法特性。

我们将对命名空间、依赖声明、模块模式以及沙箱模式进行初探——它们帮助更好地组织应用程序的代码，有效地减轻全局污染的问题。除此之外，还会对包括：私有和特权成员、静态和私有静态成员、对象常量、链以及类式函数定义方式在内的话题进行讨论。

命名空间模式（Namespace Pattern）

命名空间可以帮助减少全局变量的数量，与此同时，还能有效地避免命名冲突、名称前缀的滥用。

JavaScript默认语法并不支持命名空间，但很容易可以实现此特性。为了避免产生全局污染，你可以为应用或者类库创建一个（通常就一个）全局对象，然后将所有的功能都添加到这个对象上，而不是到处申明大量的全局函数、全局对象以及其他全局变量。

看如下例子：

```
// BEFORE: 5 globals
// Warning: antipattern
// constructors
function Parent() {}
function Child() {}
// a variable
var some_var = 1;

// some objects
var module1 = {};
module1.data = {a: 1, b: 2};
var module2 = {};
```

可以通过创建一个全局对象（通常代表应用名）来重构上述这类代码，比方说， MYAPP，然后将上述例子中的函数和变量都变为该全局对象的属性：

```
// AFTER: 1 global
// global object
var MYAPP = {};

// constructors
MYAPP.Parent = function () {};
MYAPP.Child = function () {};

// a variable
MYAPP.some_var = 1;

// an object container
MYAPP.modules = {};

// nested objects
MYAPP.modules.module1 = {};
MYAPP.modules.module1.data = {a: 1, b: 2};
MYAPP.modules.module2 = {};
```

这里的MYAPP就是命名空间对象，对象名可以随便取，可以是应用名、类库名、域名或者是公司名都可以。开发者经常约定全局变量都采用大写（所有字母都大写），这样可以显得比较突出（不过，要记住，一般大写的变量都用于表示常量）。

这种模式是一种很好的提供命名空间的方式，避免了自身代码的命名冲突，同时还避免了同一个页面上自身代码和第三方代码（比如：JavaScript类库或者小部件）的冲突。这种模式在大多数情况下非常适用，但也有它的缺点：

- 代码量稍有增加；在每个函数和变量前加上这个命名空间对象的前缀，会增加代码量，增大文件大小
- 该全局实例可以被随时修改
- 命名的深度嵌套会减慢属性值的查询

本章后续要介绍的沙箱模式则可以避免这些缺点。

通用命名空间函数

随着程序复杂度的提高，代码会分置在不同的文件中以特定顺序来加载，这样一来，就不能保证你的代码一定是第一个申明命名空间或者改变量下的属性的。甚至还会发生属性覆盖的问题。所以，在创建命名空间或者添加属性的时候，最好先检查下是否存在，如下所示：

```
// unsafe
var MYAPP = {};
// better
if (typeof MYAPP === "undefined") {
    var MYAPP = {};
}
// or shorter
var MYAPP = MYAPP || {};
```

如上所示，不难看出，如果每次做类似操作都要这样检查一下就会有很多重复性的代码。比方说，要申明**MYAPP.modules.module2**，就要重复三次这样的检查。所以，我们需要一个重用的**namespace()**函数来专门处理这些检查工作，然后用它来创建命名空间，如下所示：

```
// using a namespace function
MYAPP.namespace('MYAPP.modules.module2');

// equivalent to:
// var MYAPP = {
//    modules: {
//        module2: {}
//    }
// };
```

下面是上述namespace函数的实现案例。这种实现是无损的，意味着如果要创建的命名空间已经存在，则不会再重复创建：

```
var MYAPP = MYAPP || {};
MYAPP.namespace = function (ns_string) {
    var parts = ns_string.split('.'),
        parent = MYAPP,
        i;

    // strip redundant leading global
    if (parts[0] === "MYAPP") {
        parts = parts.slice(1);
    }

    for (i = 0; i < parts.length; i += 1) {
        // create a property if it doesn't exist
        if (typeof parent[parts[i]] === "undefined") {
            parent[parts[i]] = {};
        }
        parent = parent[parts[i]];
    }
    return parent;
};
```

上述实现支持如下使用：

```
// assign returned value to a local var
var module2 = MYAPP.namespace('MYAPP.modules.module2');
module2 === MYAPP.modules.module2; // true

// skip initial `MYAPP`
MYAPP.namespace('modules.module51');

// long namespace
MYAPP.namespace('once.upon.a.time.there.was.this.long.nested.property');
```

图5-1 展示了上述代码创建的命名空间对象在Firebug下的可视结果

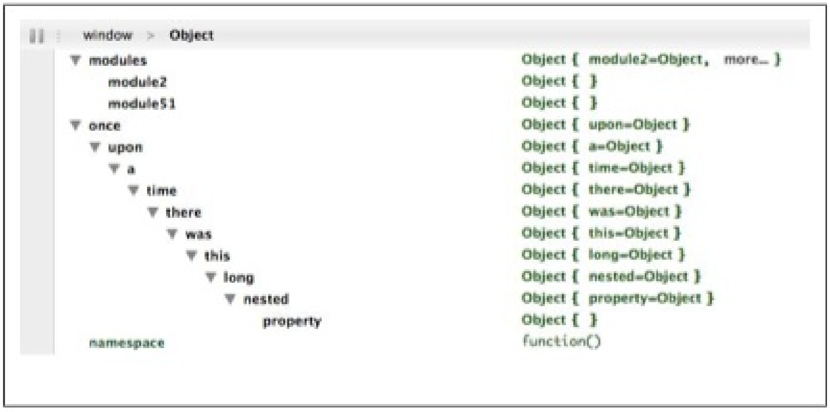


图5-1 MYAPP命名空间在Firebug下的可视结果

声明依赖

JavaScript库往往是模块化而且有用到命名空间的，这使用你可以只使用你需要的模块。比如在YUI2中，全局变量YAHOO就是一个命名空间，各个模块作为全局变量的属性，比如YAHOO.util.Dom（DOM模块）、YAHOO.util.Event（事件模块）。

将你的代码依赖在函数或者模块的顶部进行声明是一个好主意。声明就是创建一个本地变量，指向你需要用到的模块：

```
var myFunction = function () {
    // dependencies
    var event = YAHOO.util.Event,
        dom = YAHOO.util.Dom;

    // use event and dom variables
    // for the rest of the function...
};
```

这是一个相当简单的模式，但是有很多的好处：

- 明确的声明依赖是告知你代码的用户，需要保证指定的脚本文件被包含在页面中。
- 将声明放在函数顶部使得依赖很容易被查找和解析。
- 本地变量（如dom）永远会比全局变量（如YAHOO）要快，甚至比全局变量的属性（如YAHOO.util.Dom）还要快，这样会有更好的性能。使用了依赖声明模式之后，全局变量的解析在函数中只会进行一次，在此之后将会使用更快的本地变量。
- 一些高级的代码压缩工具比如YUI Compressor和Google Closure compiler会重命名本地变量（比如event可能会被压缩成一个字母，如A），这会使代码更精简，但这个操作不会对全局变量进行，因为这样做不安全。

下面的代码片段是关于是否使用依赖声明模式对压缩影响的展示。尽管使用了依赖声明模式的test2()看起来复杂，因为需要更多的代码行数和一个额外的变量，但在压缩后它的代码量却会更小，意味着用户只需要下载更少的代码：

```
function test1() {
    alert(MYAPP.modules.m1);
    alert(MYAPP.modules.m2);
    alert(MYAPP.modules.m51);
}

/*
minified test1 body:
alert(MYAPP.modules.m1);alert(MYAPP.modules.m2);alert(MYAPP.modules.m51)
*/

function test2() {
    var modules = MYAPP.modules;
    alert(modules.m1);
    alert(modules.m2);
    alert(modules.m51);
}

/*
minified test2 body:
var a=MYAPP.modules;alert(a.m1);alert(a.m2);alert(a.m51)
*/
```

私有属性和方法

JavaScript不像Java或者其它语言，它没有专门的提供私有、保护、公有属性和方法的语法。所有的对象成员都是公有的：

```
var myobj = {
    myprop: 1,
    getProp: function () {
        return this.myprop;
    }
};
console.log(myobj.myprop); // `myprop` is publicly accessible console.log(myobj.getProp()
```

当你使用构造函数创建对象的时候也是一样的，所有的成员都是公有的：

```
function Gadget() {
    this.name = 'iPod';
    this.stretch = function () {
        return 'iPad';
    };
}
var toy = new Gadget();
console.log(toy.name); // `name` is public console.log(toy.stretch()); // stretch() is pu
```

私有成员

尽管语言并没有用于私有成员的专门语法，但你可以通过闭包来实现。在构造函数中创建一个闭包，任何在这个闭包中的部分都不会暴露到构造函数之外。但是，这些私有变量却可以被公有方法访问，也就是在构造函数中定义的并且作为返回对象一部分的那些方法。我们来看一个例子，name是一个私有成员，在构造函数之外不能被访问：

```
function Gadget() {
    // private member
    var name = 'iPod';
    // public function
    this.getName = function () {
        return name;
    };
}
var toy = new Gadget();

// `name` is undefined, it's private
console.log(toy.name); // undefined
// public method has access to `name`
console.log(toy.getName()); // "iPod"
```

如你所见，在JavaScript创建私有成员很容易。你需要做的只是将私有成员放在一个函数中，保证它是函数的本地变量，也就是说让它在函数之外不可以被访问。

特权方法

特权方法的概念不涉及到任何语法，它只是一个给可以访问到私有成员的公有方法的名字（就像它们有更多权限一样）。

在前面的例子中，getName()就是一个特权方法，因为它有访问name属性的特殊权限。

私有成员失效

当你使用私有成员时，需要考虑一些极端情况：

- 在Firefox的一些早期版本中，允许通过给eval()传递第二个参数的方法来指定上下文对象，从而允许访问函数的私有作用域。比如在Mozilla Rhino（译注：一个JavaScript引擎）中，允许使用 `__parent__` 来访问私有作用域。现在这些极端情况并没有被广泛应用到浏览器中。
- 当你直接通过特权方法返回一个私有变量，而这个私有变量恰好是一个对象或者数组时，外部的代码可以修改这个私有变量，因为它是按引用传递的。

我们来看一下第二种情况。下面的Gadget的实现看起来没有问题：

```
function Gadget() {
    // private member
    var specs = {
    screen_width: 320,
    screen_height: 480,
    color: "white"
    };

    // public function
    this.getSpecs = function () {
        return specs;
    };
}
```

这里的问题是getSpecs()返回了一个specs对象的引用。这使得Gadget的使用者可以修改貌似隐藏起来的私有成员specs：

```
var toy = new Gadget(),
    specs = toy.getSpecs();

specs.color = "black";
specs.price = "free";

console.dir(toy.getSpecs());
```

在Firebug控制台中打印出来的结果如图5-2：

图5-2 私有对象被修改了

这个意外的问题的解决方法就是不要将你想保持私有的对象或者数组的引用传递出去。达到这个目标的一种方法是让getSpecs()返回一个新对象，这个新对象只包含对象的使用者感兴趣的数据。这也是众所周知的“最低授权原则”（Principle of Least Authority，简称POLA），指永远不要给出比需求更多的东西。在这个例子中，如果Gadget的使用者关注它是否适应一个特定的盒子，它只需要知道尺寸即可。所以你应该创建一个getDimensions()，用它返回一个只包含width和height的新对象，而不是把什么都给出去。也就是说，也许你根本不需要实现getSpecs()方法。

当你需要传递所有的数据时，有另外一种方法，就是使用通用的对象复制函数创建specs对象的一个副本。下一章提供了两个这样的函数——一个叫extend()，它会浅复制一个给定的对象（只复制顶层的成员）。另一个叫extendDeep()，它会做深复制，遍历所有的属性和嵌套的属性。

对象字面量和私有成员

到目前为止，我们只看了使用构建函数创建私有成员的示例。如果使用对象字面量创建对象时会是什么情况呢？是否有可能含有私有成员？

如你前面所看到的那样，私有数据使用一个函数来包裹。所以在使用对象字面量时，你也可以使用一个立即执行的匿名函数创建的闭包。例如：

```
var myobj; // this will be the object
(function () {
    // private members
    var name = "my, oh my";

    // implement the public part
    // note -- no `var`
    myobj = {
        // privileged method
        getName: function () {
            return name;
        }
    };
}());

myobj.getName(); // "my, oh my"
```

还有一个原理一样但看起来不一样的实现示例：

```
var myobj = (function () {
    // private members
    var name = "my, oh my";

    // implement the public part
    return {
        getName: function () {
            return name;
        }
    };
}());

myobj.getName(); // "my, oh my"
```

这个例子也是所谓的“模块模式”的基础，我们稍后将讲到它。

原型和私有成员

使用构造函数创建私有成员的一个弊端是，每一次调用构造函数创建对象时这些私有成员都会被创建一次。

这对在构建函数中添加到 `this` 的成员来说是一个问题。为了避免重复劳动，节省内存，你可以将共用的属性和方法添加到构造函数的 `prototype`（原型）属性中。这样的话这些公共的部分会在使用同一个构造函数创建的所有实例中共享。你也同样可以在这些实例中共享私有成员。你可以将两种模式联合起来达到这个目的：构造函数中的私有属性和对象字面量中的私有属性。因为 `prototype` 属性也只是一个对象，可以使用对象字面量创建。

这是一个示例：

```
function Gadget() {
    // private member
    var name = 'iPod';
    // public function
    this.getName = function () {
        return name;
    };
}

Gadget.prototype = (function () {
    // private member
    var browser = "Mobile Webkit";
    // public prototype members
    return {
        getBrowser: function () {
            return browser;
        }
    };
}());

var toy = new Gadget();
console.log(toy.getName()); // privileged "own" method console.log(toy.getBrowser()); //
```

将私有函数暴露为公有方法

“暴露模式”是指将已经有的私有函数暴露为公有方法。当对对象进行操作时，所有功能代码都对这些操作很敏感，而你想尽量保护这些代码的时候很有用。（译注：指对来自外部的修改很敏感。）但同时，你又希望能提供一些功能的访问权限，因为它们会被用到。如果你把这些方法公开，就会使得它们不再健壮，你的API的使用者可能修改它们。在ECMAScript5中，你可以选择冻结一个对象，但在之前的版本中不可用。下面进入暴露模式（原来是由Christian Heilmann创造的模式，叫“暴露模块模式”）。

我们来看一个例子，它建立在对象字面量的私有成员模式之上：

```
var myarray;

(function () {

    var astr = "[object Array]",
        toString = Object.prototype.toString;

    function isArray(a) {
        return toString.call(a) === astr;
    }

    function indexOf(haystack, needle) {
        var i = 0,
            max = haystack.length;
        for (; i < max; i += 1) {
            if (haystack[i] === needle) {
                return i;
            }
        }
        return −1;
    }

    myarray = {
        isArray: isArray,
        indexOf: indexOf,
        inArray: indexOf
    };

}());
```

这里有两个私有变量和两个私有函数—— `isArray()` 和 `indexOf()` 。在包裹函数的最后，使用那些允许被从外部访问的函数填充 `myarray` 对象。在这个例子中，同一个私有函数 `indexOf()` 同时被暴露为ECMAScript 5风格的 `indexOf` 和PHP风格的 `inArry` 。测试一下myarray对象：

```
myarray.isArray([1,2]); // true
myarray.isArray({0: 1}); // false
myarray.indexOf(["a", "b", "z"], "z"); // 2
myarray.inArray(["a", "b", "z"], "z"); // 2
```

现在假如有一些意外的情况发生在暴露的 `indexOf()` 方法上，私有的 `indexOf()` 方法仍然是安全的，因此 `inArray()` 仍然可以正常工作：

```
myarray.indexOf = null;
myarray.inArray(["a", "b", "z"], "z"); // 2
```

模块模式

模块模式使用得很广泛，因为它可以为代码提供特定的结构，帮助组织日益增长的代码。不像其它语言，JavaScript没有专门的“包”（package）的语法，但模块模式提供了用于创建独立解耦的代码片段的工具，这些代码可以被当成黑盒，当你正在写的软件需求发生变化时，这些代码可以被添加、替换、移除。

模块模式是我们目前讨论过的好几种模式的组合，即：

- 命名空间模式
- 立即执行的函数模式
- 私有和特权成员模式
- 依赖声明模式

第一步是初始化一个命名空间。我们使用本章前面部分的 `namespace()` 函数，创建一个提供数组相关方法的套件模块：

```
MYAPP.namespace('MYAPP.utilities.array');
```

下一步是定义模块。使用一个立即执行的函数来提供私有作用域供私有成员使用。立即执行的函数返回一个对象，也就是带有公有接口的真正的模块，可以供其它代码使用：

```
MYAPP.utilities.array = (function () {
    return {
        // todo...
    };
}());
```

下一步，给公有接口添加一些方法：

```
MYAPP.utilities.array = (function () {
    return {
        inArray: function (needle, haystack) {
            // ...
        },
        isArray: function (a) {
            // ...
        }
    };
}());
```

如果需要的话，你可以在立即执行的函数提供的闭包中声明私有属性和私有方法。函数顶部也是声明依赖的地方。在变量声明的下方，你可以选择性地放置辅助初始化模块的一次性代码。函数最终返回的是一个包含模块公共API的对象：

MYAPP.namespace('MYAPP.utilities.array'); MYAPP.utilities.array = (function () {

```
    // dependencies
var uobj = MYAPP.utilities.object,
    ulang = MYAPP.utilities.lang,

    // private properties
    array_string = "[object Array]",
    ops = Object.prototype.toString;

    // private methods
    // ...
    // end var

// optionally one-time init procedures
// ...

// public API
return {

    inArray: function (needle, haystack) {
        for (var i = 0, max = haystack.length; i < max; i += 1) {
            if (haystack[i] === needle) {
                return true;
            }
        }
    },

    isArray: function (a) {
        return ops.call(a) === array_string;
    }
    // ... more methods and properties
};
```

}());

模块模式被广泛使用，这是一种值得强烈推荐的模式，它可以帮助组织代码，尤其是代码量在不断增长的时候。

暴露模块模式

我们在本章中讨论私有成员模式时已经讨论过暴露模式。模块模式也可以用类似的方法来组织，将所有的方法保持私有，只在最后暴露需要使用的方法来初始化API。

上面的例子可以变成这样：

```
MYAPP.utilities.array = (function () {

        // private properties
    var array_string = "[object Array]",
        ops = Object.prototype.toString,

        // private methods
        inArray = function (haystack, needle) {
            for (var i = 0, max = haystack.length; i < max; i += 1) {
                if (haystack[i] === needle) {
                    return i;
                }
            }
            return −1;
        },
        isArray = function (a) {
            return ops.call(a) === array_string;
        };
        // end var

    // revealing public API
    return {
        isArray: isArray,
        indexOf: inArray
    };
}());
```

创建构造函数的模块

前面的例子创建了一个对象 `MYAPP.utilities.array` ，但有时候使用构造函数来创建对象会更方便。你也可以同样使用模块模式来做。唯一的区别是包裹模块的立即执行的函数会在最后返回一个函数，而不是一个对象。

看下面的模块模式的例子，创建了一个构造函数 `MYAPP.utilities.Array` ：

```
MYAPP.namespace('MYAPP.utilities.Array');

MYAPP.utilities.Array = (function () {

        // dependencies
    var uobj = MYAPP.utilities.object,
        ulang = MYAPP.utilities.lang,

        // private properties and methods...
        Constr;

        // end var

        // optionally one-time init procedures
        // ...

        // public API -- constructor
        Constr = function (o) {
            this.elements = this.toArray(o);
        };
        // public API -- prototype
        Constr.prototype = {
            constructor: MYAPP.utilities.Array,
            version: "2.0",
            toArray: function (obj) {
                for (var i = 0, a = [], len = obj.length; i < len; i += 1) {
                    a[i] = obj[i];
                }
                return a;
            }
        };

    // return the constructor
    // to be assigned to the new namespace return Constr;

}());
```

像这样使用这个新的构造函数：

```
var arr = new MYAPP.utilities.Array(obj);
```

在模块中引入全局上下文

作为这种模式的一个常见的变种，你可以给包裹模块的立即执行的函数传递参数。你可以传递任何值，但通常会传递全局变量甚至是全局对象本身。引入全局上下文可以加快函数内部的全局变量的解析，因为引入之后会作为函数的本地变量：

```
MYAPP.utilities.module = (function (app, global) {

    // references to the global object
    // and to the global app namespace object
    // are now localized

}(MYAPP, this));
```

沙箱模式

沙箱模式主要着眼于命名空间模式的短处，即：

- 依赖一个全局变量成为应用的全局命名空间。在命名空间模式中，没有办法在同一个页面中运行同一个应用或者类库的不同版本，在为它们都会需要同一个全局变量名，比如 `MYAPP` 。
- 代码中以点分隔的名字比较长，无论写代码还是解析都需要处理这个很长的名字，比如 `MYAPP.utilities.array` 。

顾名思义，沙箱模式为模块提供了一个环境，模块在这个环境中的任何行为都不会影响其它的模块和其它模块的沙箱。

这个模式在YUI3中用得很多，但是需要记住的是，下面的讨论只是一些示例实现，并不讨论YUI3中的消息箱是如何实现的。

全局构造函数

在命名空间模式中 ，有一个全局对象，而在沙箱模式中，唯一的全局变量是一个构造函数，我们把它命名为 `Sandbox()` 。我们使用这个构造函数来创建对象，同时也要传入一个回调函数，这个函数会成为代码运行的独立空间。

使用沙箱模式是像这样：

```
new Sandbox(function (box) {
    // your code here...
});
```

`box` 对象和命名空间模式中的 `MYAPP` 类似，它包含了所有你的代码需要用到的功能。

我们要多做两件事情：

- 通过一些手段（第3章中的强制使用new的模式），你可以在创建对象的时候不要求一定有new。
- 让 `Sandbox()` 构造函数可以接受一个（或多个）额外的配置参数，用于指定这个对象需要用到的模块名字。我们希望代码是模块化的，因此绝大部分 `Sandbox()` 提供的功能都会被包含在模块中。

有了这两个额外的特性之后，我们来看一下实例化对象的代码是什么样子。

你可以在创建对象时省略 `new` 并像这样使用已有的“ajax”和“event”模块：

```
Sandbox(['ajax', 'event'], function (box) {
    // console.log(box);
});
```

下面的例子和前面的很像，但是模块名字是作为独立的参数传入的：

```
Sandbox('ajax', 'dom', function (box) {
    // console.log(box);
});
```

使用通配符“*”来表示“使用所有可用的模块”如何？为了方便，我们也假设没有任何模块传入时，沙箱使用“*”。所以有两种使用所有可用模块的方法：

```
Sandbox('*', function (box) {
    // console.log(box);
});

Sandbox(function (box) {
    // console.log(box);
});
```

下面的例子展示了如何实例化多个消息箱对象，你甚至可以将它们嵌套起来而互不影响：

```
Sandbox('dom', 'event', function (box) {

    // work with dom and event

    Sandbox('ajax', function (box) {
        // another sandboxed "box" object
        // this "box" is not the same as
        // the "box" outside this function

        //...

        // done with Ajax
    });

    // no trace of Ajax module here
});
```

从这些例子中看到，使用沙箱模式可以通过将代码包裹在回调函数中的方式来保护全局命名空间。

如果需要的话，你也可以利用函数也是对象这一事实，将一些数据作为静态属性存放到`Sandbox()`构造函数。

最后，你可以根据需要的模块类型创建不同的实例，这些实例都是相互独立的。

现在我们来看一下如何实现`Sandbox()`构造函数和它的模块来支持上面讲到的所有功能。

添加模块

在动手实现构造函数之前，我们来看一下如何添加模块。

`Sandbox()`构造函数也是一个对象，所以可以给它添加一个`modules`静态属性。这个属性也是一个包含名值（key-value）对的对象，其中key是模块的名字，value是模块的功能实现。

```
Sandbox.modules = {};

Sandbox.modules.dom = function (box) {
    box.getElement = function () {};
    box.getStyle = function () {};
    box.foo = "bar";
};

Sandbox.modules.event = function (box) {
    // access to the Sandbox prototype if needed:
    // box.constructor.prototype.m = "mmm";
    box.attachEvent = function () {};
    box.dettachEvent = function () {};
};

Sandbox.modules.ajax = function (box) {
    box.makeRequest = function () {};
    box.getResponse = function () {};
};
```

在这个例子中我们添加了`dom`、`event`和`ajax`模块，这些都是在每个类库或者复杂的web应用中很常见的代码片段。

实现每个模块功能的函数接受一个实例`box`作为参数，并给这个实例添加属性和方法。

实现构造函数

最后，我们来实现`Sandbox()`构造函数（你可能会很自然地想将这类构造函数命名为对你的类库或者应用有意义的名字）：

```
function Sandbox() {
        // turning arguments into an array
    var args = Array.prototype.slice.call(arguments),
        // the last argument is the callback
        callback = args.pop(),
        // modules can be passed as an array or as individual parameters
        modules = (args[0] && typeof args[0] === "string") ? args : args[0], i;

    // make sure the function is called
    // as a constructor
    if (!(this instanceof Sandbox)) {
        return new Sandbox(modules, callback);
    }

    // add properties to `this` as needed:
    this.a = 1;
    this.b = 2;

    // now add modules to the core `this` object
    // no modules or "*" both mean "use all modules"
    if (!modules || modules === '*') {
        modules = [];
        for (i in Sandbox.modules) {
            if (Sandbox.modules.hasOwnProperty(i)) {
                modules.push(i);
            }
        }
    }

    // initialize the required modules
    for (i = 0; i < modules.length; i += 1) {
        Sandbox.modules[modules[i]](this);
    }

    // call the callback
    callback(this);
}

// any prototype properties as needed
Sandbox.prototype = {
    name: "My Application",
    version: "1.0",
    getName: function () {
        return this.name;
    }
};
```

这个实现中的一些关键点：

- 有一个检查`this`是否是`Sandbox`实例的过程，如果不是（也就是调用`Sandbox()`时没有加`new`），我们将这个函数作为构造函数再调用一次。
- 你可以在构造函数中给`this`添加属性，也可以给构造函数的原型添加属性。
- 被依赖的模块可以以数组的形式传递，也可以作为单独的参数传递，甚至以`*`通配符（或者省略）来表示加载所有可用的模块。值得注意的是，我们在这个示例实现中并没有考虑从外部文件中加载模块，但明显这是一个值得考虑的事情。比如YUI3就支持这种情况，你可以只加载最基本的模块（作为“种子”），其余需要的任何模块都通过将模块名和文件名对应的方式从外部文件中加载。
- 当我们知道依赖的模块之后就初始化它们，也就是调用实现每个模块的函数。
- 构造函数的最后一个参数是回调函数。这个回调函数会在最后使用新创建的实例来调用。事实上这个回调函数就是用户的沙箱，它被传入一个`box`对象，这个对象包含了所

有依赖的功能。

静态成员

静态属性和方法是指那些在所有的实例中保持一致的成员。在基于类的语言中，表态成员是用专门的语法来创建，使用时就像是类自己的成员一样。比如`MathUtils`类的`max()`方法会被像这样调用：`MathUtils.max(3, 5)`。这是一个公有静态成员的示例，即可以在不实例化类的情况下使用。同样也可以有私有的静态方法，即对类的使用者不可见，而在类的所有实例间是共享的。我们来看一下如何在JavaScript中实现公有和私有静态成员。

公有静态成员

在JavaScript中没有专门用于静态成员的语法。但通过给构造函数添加属性的方法，可以拥有和基于类的语言一样的使用语法。之所有可以这样做是因为构造函数和其它的函数一样，也是对象，可以拥有属性。前一章讨论过的Memoization模式也使用了同样的方法，即给函数添加属性。

下面的例子定义了一个构造函数`Gadget`，它有一个静态方法`isShiny()`和一个实例方法`setPrice()`。`isShiny()`是一个静态方法，因为它不需要指定一个对象才能工作（就像你不需要先指定一个工具（gadget）才知道所有的工具是不是有光泽的（shiny））。但setPrice()却需要一个对象，因为工具可能有不同的定价：

```
// constructor
var Gadget = function () {};

// a static method
Gadget.isShiny = function () {
    return "you bet";
};

// a normal method added to the prototype Gadget.prototype.setPrice = function (price) {
    this.price = price;
};
```

现在我们来调用这些方法。静态方法`isShiny()`可以直接在构造函数上调用，但其它的常规方法需要一个实例：

```
// calling a static method
Gadget.isShiny(); // "you bet"

// creating an instance and calling a method
var iphone = new Gadget();
iphone.setPrice(500);
```

使用静态方法的调用方式去调用实例方法并不能正常工作，同样，用调用实例方法的方式来调用静态方法也不能正常工作：

```
typeof Gadget.setPrice; // "undefined"
typeof iphone.isShiny; // "undefined"
```

有时候让静态方法也能用在实例上会很方便。我们可以通过在原型上加一个新方法来很容易地做到这点，这个新方法作为原来的静态方法的一个包装：

```
Gadget.prototype.isShiny = Gadget.isShiny;
iphone.isShiny(); // "you bet"
```

在这种情况下，你需要很小心地处理静态方法内的 `this` 。当你运行 `Gadget.isShiny()` 时，在 `isShiny()` 内部的 `this` 指向 `Gadget` 构造函数。而如果你运行 `iphone.isShiny()` ，那么 `this` 会指向 `iphone` 。

最后一个例子展示了同一个方法被静态调用和非静态调用时明显不同的行为，这取决于调用的方式。这里的 `instanceof` 用于获方法是如何被调用的：

```
// constructor
var Gadget = function (price) {
    this.price = price;
};

// a static method
Gadget.isShiny = function () {

    // this always works
    var msg = "you bet";

    if (this instanceof Gadget) {
        // this only works if called non-statically
        msg += ", it costs $" + this.price + '!';
    }

    return msg;
};

// a normal method added to the prototype
Gadget.prototype.isShiny = function () {
    return Gadget.isShiny.call(this);
};
```

测试一下静态方法调用：

```
Gadget.isShiny(); // "you bet"
```

测试一下实例中的非静态调用：

```
var a = new Gadget('499.99');
a.isShiny(); // "you bet, it costs $499.99!"
```

私有静态成员

到目前为止，我们都只讨论了公有的静态方法，现在我们来看一下如何实现私有静态成员。所谓私有静态成员是指：

- 被所有由同一构造函数创建的对象共享
- 不允许在构造函数外部访问

我们来看一个例子，`counter`是`Gadget`构造函数的一个私有静态属性。在本章中我们已经讨论过私有属性，这里的做法也是一样，需要一个函数提供的闭包来包裹私有成员。然后让这个包裹函数立即执行并返回一个新的函数。将这个返回的函数赋值给`Gadget`作为构造函数。

```
var Gadget = (function () {

    // static variable/property
    var counter = 0;

    // returning the new implementation
    // of the constructor
    return function () {
        console.log(counter += 1);
    };

}()); // execute immediately
```

这个`Gadget`构造函数只简单地增加私有的`counter`的值然后打印出来。用多个实例测试的话你会看到`counter`在实例之间是共享的：

```
var g1 = new Gadget();// logs 1
var g2 = new Gadget();// logs 2
var g3 = new Gadget();// logs 3
```

因为我们在创建每个实例的时候`counter`的值都会加1，所以它实际上成了唯一标识使用`Gadget`构造函数创建的对象的ID。这个唯一标识可能会很有用，那为什么不把它通用一个特权方法暴露出去呢？（译注：其实这里不能叫ID，只是一个记录有多少个实例的数字而已，因为如果有多个实例被创建的话，其实已经没办法取到前面实例的标识了。）下面的例子是基于前面的例子，增加了用于访问私有静态属性的`getLastId()`方法：

```
// constructor
var Gadget = (function () {

    // static variable/property
    var counter = 0,
        NewGadget;

    // this will become the
    // new constructor implementation
    NewGadget = function () {
        counter += 1;
    };

    // a privileged method
    NewGadget.prototype.getLastId = function () {
        return counter;
    };

    // overwrite the constructor
    return NewGadget;

}()); // execute immediately
```

测试这个新的实现：

```
var iphone = new Gadget();
iphone.getLastId(); // 1
var ipod = new Gadget();
ipod.getLastId(); // 2
var ipad = new Gadget();
ipad.getLastId(); // 3
```

静态属性（包括私有和公有）有时候会非常方便，它们可以包含和具体实例无关的方法和数据，而不用在每次实例中再创建一次。当我们在第七章中讨论单例模式时，你可以看到使用静态属性实现类式单例构造函数的例子。

对象常量

JavaScript中是没有常量的，尽管在一些比较现代的环境中可能会提供 `const` 来创建常量。

一种常用的解决办法是通过命名规范，让不应该变化的变量使用全大写。这个规范实际上也用在JavaScript原生对象中：

```
Math.PI; // 3.141592653589793
Math.SQRT2; // 1.4142135623730951
Number.MAX_VALUE; // 1.7976931348623157e+308
```

你自己的常量也可以用这种规范，然后将它们作为静态属性加到构造函数中：

```
// constructor
var Widget = function () {
    // implementation...
};

// constants
Widget.MAX_HEIGHT = 320;
Widget.MAX_WIDTH = 480;
```

同样的规范也适用于使用字面量创建的对象，常量会是使用大写名字的普通名字。

如果你真的希望有一个不能被改变的值，那么可以创建一个私有属性，然后提供一个取值的方法（getter），但不给赋值的方法（setter）。这种方法在很多可以用命名规范解决的情况下可能有些矫枉过正，但不失为一种选择。

下面是一个通过的`constant`对象的实现，它提供了这些方法：

- set(name, value)

  定义一个新的常量

- isDefined(name)

  检查一个常量是否存在

- get(name)

  取常量的值

在这个实现中，只允许基本类型的值成为常量。同时还要使用`hasOwnproperty()`小心地处理那些恰好是原生属性的常量名，比如`toString`或者`hasOwnProperty`，然后给所有的常量名加上一个随机生成的前缀：

```
var constant = (function () {
    var constants = {},
        ownProp = Object.prototype.hasOwnProperty,
        allowed = {
            string: 1,
            number: 1,
            boolean: 1
        },
        prefix = (Math.random() + "_").slice(2);
    return {
        set: function (name, value) {
            if (this.isDefined(name)) {
                return false;
            }
            if (!ownProp.call(allowed, typeof value)) {
                return false;
            }
            constants[prefix + name] = value;
            return true;
        },
        isDefined: function (name) {
            return ownProp.call(constants, prefix + name);
        },
        get: function (name) {
            if (this.isDefined(name)) {
                return constants[prefix + name];
            }
            return null;
        }
    };
}());
```

测试这个实现：

```
// check if defined
constant.isDefined("maxwidth"); // false

// define
constant.set("maxwidth", 480); // true

// check again
constant.isDefined("maxwidth"); // true

// attempt to redefine
constant.set("maxwidth", 320); // false

// is the value still intact?
constant.get("maxwidth"); // 480
```

链式调用模式

使用链式调用模式可以让你在一对个象上连续调用多个方法，不需要将前一个方法的返回值赋给变量，也不需要将多个方法调用分散在多行：

```
myobj.method1("hello").method2().method3("world").method4();
```

当你创建了一个没有有意义的返回值的方法时，你可以让它返回this，也就是这些方法所属的对象。这使得对象的使用者可以将下一个方法的调用和前一次调用链起来：

```
var obj = {
    value: 1,
    increment: function () {
        this.value += 1;
        return this;
    },
    add: function (v) {
        this.value += v;
        return this;
    },
    shout: function () {
        alert(this.value);
    }
};

// chain method calls
obj.increment().add(3).shout(); // 5

// as opposed to calling them one by one
obj.increment();
obj.add(3);
obj.shout(); // 5
```

链式调用模式的利弊

使用链式调用模式的一个好处就是可以节省代码量，使得代码更加简洁和易读，读起来就像在读句子一样。

另外一个好处就是帮助你思考如何拆分你的函数，创建更小、更有针对性的函数，而不是一个什么都做的函数。长时间来看，这会提升代码的可维护性。

一个弊端是调用这样写的代码会更困难。你可能知道一个错误出现在某一行，但这一行要做很多的事情。当链式调用的方法中的某一个出现问题而又没报错时，你无法知晓到底是哪一个出问题了。《代码整洁之道》的作者Robert Martion甚至叫这种模式为“train wreck”模式。（译注：直译为“火车事故”，指负面影响比较大。）

不管怎样，认识这种模式总是好的，当你写的方法没有明显的有意义的返回值时，你就可以返回 `this` 。这个模式应用得很广泛，比如jQuery库。如果你去看DOM的API的话，你会发现它也会以这样的形式倾向于链式调用：

```
document.getElementsByTagName('head')[0].appendChild(newnode);
```

method()方法

JavaScript对于习惯于用类来思考的人来说可能会比较费解，这也是很多开发者希望将JavaScript代码变得更像基于类的语言的原因。其中的一种尝试就是由Douglas Crockford提出来的 `method()` 方法。其实，他也承认将JavaScript变得像基于类的语言是不推荐的方法，但不管怎样，这都是一种有意思的模式，你可能会在一些应用中见到。

使用构造函数主须Java中使用类一样。它也允许你在构造函数体的 `this` 中添加实例属性。但是在 `this` 中添加方法却是不高效的，因为最终这些方法会在每个实例中被重新创建一次，这样会花费更多的内存。这也是为什么可重用的方法应该被放到构造函数的 `prototype` 属性（原型）中的原因。但对很多开发者来说，`prototype` 可能跟个外星人一样陌生，所以你可以通过一个方法将它隐藏起来。

> 给语言添加一个使用起来更方便的方法一般叫作“语法糖”。在这个例子中，你可以将 `method()` 方法称为一个语法糖方法。

使用这个语法糖方法 `method()` 来定义一个“类”是像这样：

```
var Person = function (name) {
    this.name = name;
}.
    method('getName', function () {
        return this.name;
    }).
    method('setName', function (name) {
        this.name = name;
        return this;
    });
```

注意构造函数和调用 `method()` 是如何链起来的，接下来又链式调用了下一个 `method()` 方法。这就是我们前面讨论的链式调用模式，可以帮助我们用一个语句完成对整个“类”的定义。

`method()` 方法接受两个参数：

- 新方法的名字
- 新方法的实现

然后这个新方法被添加到 `Person` “类”。新方法的实现也只是一个函数，在这个函数里面 `this` 指向由 `Person` 创建的对象，正如我们期望的那样。

下面是使用 `Person()` 创建和使用新对象的代码：

```
var a = new Person('Adam');
a.getName(); // 'Adam'
a.setName('Eve').getName(); // 'Eve'
```

同样地注意链式调用，因为 `setName()` 返回了 `this` 就可以链式调用了。

最后是 `method()` 方法的实现：

```
if (typeof Function.prototype.method !== "function") {
    Function.prototype.method = function (name, implementation) {
        this.prototype[name] = implementation;
        return this;
    };
}
```

在 `method()` 的实现中，我们首先检查这个方法是否已经被实现过，如果没有则继续，将传入的参数 `implementation` 加到构造函数的原型中。在这里 `this` 指向构造函数，而我们要增加的功能正在在这个构造函数的原型上。

小结

在本章中你看到了好几种除了字面量和构造函数之外的创建对象的方法。

你看到了使用命名空间模式来保持全局空间干净和帮助组织代码。看到了简单而又有用的依赖声明模式。然后我们详细讨论了有关私有成员的模式，包括私有成员、特权方法以及一些涉及私有成员的极端情况，还有使用对象字面量创建私有成员以及将私有方法暴露为公有方法。所有这些模式都是搭建起现在流行而强大的模块模式的积木。

然后你看到了使用沙箱模式作为长命名空间的另一种选择，它可以为你的代码和模块提供独立的环境。

在最后，我们深入讨论了对象常量、静态成员（公有和私有）、链式调用模式，以及神奇的 `method()` 方法。

代码复用模式

代码复用是一个既重要又有趣的话题，因为努力在自己或者别人写的代码上写尽量少且可以复用的代码是件很自然的事情，尤其当这些代码是经过测试的、可维护的、可扩展的、有文档的时候。

当我们说到代码复用的时候，想到的第一件事就是继承，本章会有很大篇幅讲述这个话题。你将看到好多种方法来实现“类式（classical）”和一些其它方式的继承。但是，最最重要的事情，是你需要记住终极目标——代码复用。继承是达到这个目标的一种方法，但是不是唯一的。在本章，你将看到怎样基于其它对象来构建新对象，怎样使用混元，以及怎样在不使用继承的情况下只复用你需要的功能。

在做代码复用的工作的时候，谨记Gang of Four 在书中给出的关于对象创建的建议：“优先使用对象创建而不是类继承”。（译注：《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software）是一本设计模式的经典书籍，该书作者为Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides，被称为“Gang of Four”，简称“GoF”。）

类式继承 vs 现代继承模式

在讨论JavaScript的继承这个话题的时候，经常会听到“类式继承”的概念，那我们先看一下什么是类式（classical）继承。classical一词并不是来自某些古老的、固定的或者是被广泛接受的解决方案，而仅仅是来自单词“class”。（译注：classical也有“经典”的意思。）

很多编程语言都有原生的类的概念，作为对象的蓝本。在这些语言中，每个对象都是一个指定类的实例（instance），并且（以Java为例）一个对象不能在不存在对应的类的情况下存在。在JavaScript中，因为没有类，所以类的实例的概念没什么意义。JavaScript的对象仅仅是简单的键值对，这些键值对都可以动态创建或者是改变。

但是JavaScript拥有构造函数（constructor functions），并且有语法和使用类非常相似的new运算符。

在Java中你可能会这样写：

```
Person adam = new Person();
```

在JavaScript中你可以这样：

```
var adam = new Person();
```

除了Java是强类型语言需要给adam添加类型Person外，其它的语法看起来是一样的。JavaScript的创建函数调用看起来感觉Person是一个类，但事实上，Person仅仅是一个函数。语法上的相似使得非常多的开发者陷入对JavaScript类的思考，并且给出了很多模拟类的继承方案。这样的实现方式，我们叫它"类式继承"。顺便也提一下，所谓"现代"继承模式是指那些不需要你去想类这个概念的模式。

当需要给项目选择一个继承模式时，有不少的备选方案。你应该尽量选择那些现代继承模式，除非团队已经觉得"无类不欢"。

本章先讨论类式继承，然后再关注现代继承模式。

类式继承的期望结果

实现类式继承的目标是基于构造函数Child()来创建一个对象，然后从另一个构造函数Parent()获得属性。

> 尽管我们是在讨论类式继承，但还是尽量避免使用"类"这个词。"构造函数"或者"constructor"虽然更长，但是更准确，不会让人迷惑。通常情况下，应该努力避免在跟团队沟通的时候使用"类"这个词，因为在JavaScript中，很可能每个人都会有不同的理解。

下面是定义两个构造函数Parent()和Child()的例子：

```
//parent构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

//给原型增加方法
Parent.prototype.say = function () {
    return this.name;
};

//空的child构造函数
function Child(name) {}

//继承
inherit(Child, Parent);
```

上面的代码定义了两个构造函数Parent()和Child()，say()方法被添加到了Parent()构建函数的原型（prototype）中，inherit()函数完成了继承的工作。inherit()函数并不是原生提供的，需要自己实现。让我们来看一看比较大众的实现它的几种方法。

类式继承1——默认模式

最常用的一种模式是使用Parent()构造函数来创建一个对象，然后把这个对象设为Child()的原型。这是可复用的inherit()函数的第一种实现方法：

```
function inherit(C, P) {
    C.prototype = new P();
}
```

需要强调的是原型（prototype属性）应该指向一个对象，而不是函数，所以它需要指向由父构造函数创建的实例（对象），而不是构造函数自己。换句话说，请注意new运算符，有了它这种模式才可以正常工作。

之后在应用中使用new Child()创建对象的时候，它将通过原型拥有Parent()实例的功能，像下面的例子一样：

```
var kid = new Child();
kid.say(); // "Adam"
```

跟踪原型链

在这种模式中，子对象既继承了（父对象）“自己的属性”（添加给this的实例属性，比如name），也继承了原型中的属性和方法（比如say()）。

我们来看一下在这种继承模式中原型链是怎么工作的。为了讨论方便，我们假设对象是内在中的一块空间，它包含数据和指向其它空间的引用。当使用new Parent()创建一个对象时，这样的一块空间就被分配了（图6-1中的2号）。它保存着name属性的数据。如果你尝试访问say()方法（比如通过(new Parent).say()），2号空间中并没有这个方法。但是在通过隐藏的链接**proto**指向Parent()构建函数的原型prototype属性时，就可以访问到包含say()方法的1号空间（Parent.prototype）了。所有的这一块都是在幕后发生的，不需要任何额外的操作，但是知道它是怎样工作的以及你正在访问或者修正的数据在哪是很重要的。注意，**proto**在这里只是为了解释原型链，这个属性在语言本身中是不可用的，尽管有一些环境提供了（比如Firefox）。

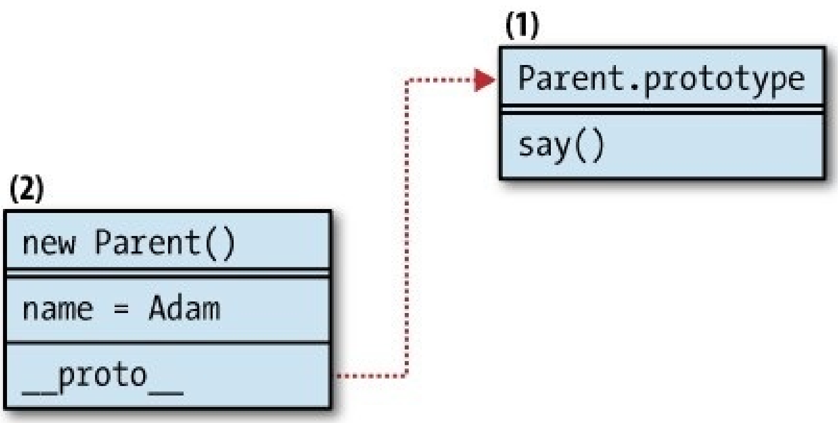


图6-1 Parent()构造函数的原型链

现在我们来看一下在使用inherit()函数之后再使用var kid = new Child()创建一个新对象时会发生什么。见图6-2。

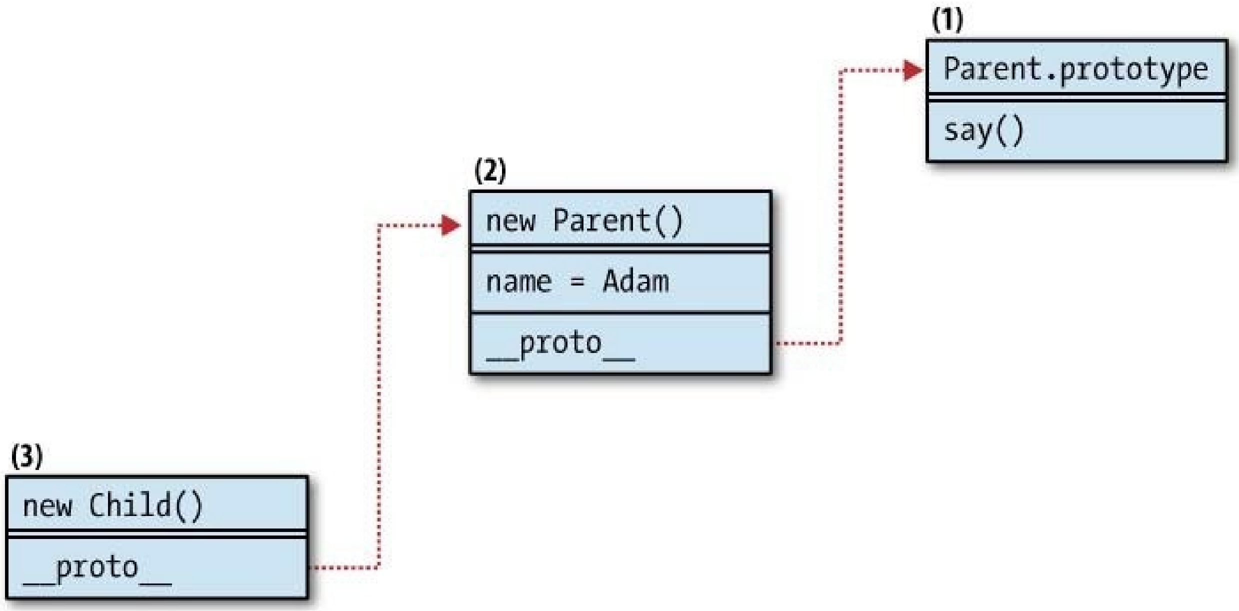


图6-2 继承后的原型链

Child()构造函数是空的，也没有属性添加到Child.prototype上，这样，使用new Child()创建出来的对象都是空的，除了有隐藏的链接**proto**。在这个例子中，**proto**指向在inherit()函数中创建的new Parent()对象。

现在使用kid.say()时会发生什么？3号对象没有这个方法，所以通过原型链找到2号。2号对象也没有这个方法，所以也通过原型链找到1号，刚好有这个方法。接下来say()方法引用了this.name，这个变量也需要解析。于是沿原型链查找的过程又走了一遍。在这个例子中，this指向3号对象，它没有name属性。然后2号对象被访问，并且有name属性，值为“Adam”。

最后，我们多看一点东西，假如我们有如下的代码：

```
var kid = new Child();
kid.name = "Patrick";
kid.say(); // "Patrick"
```

图6-3展现了这个例子的原型链：

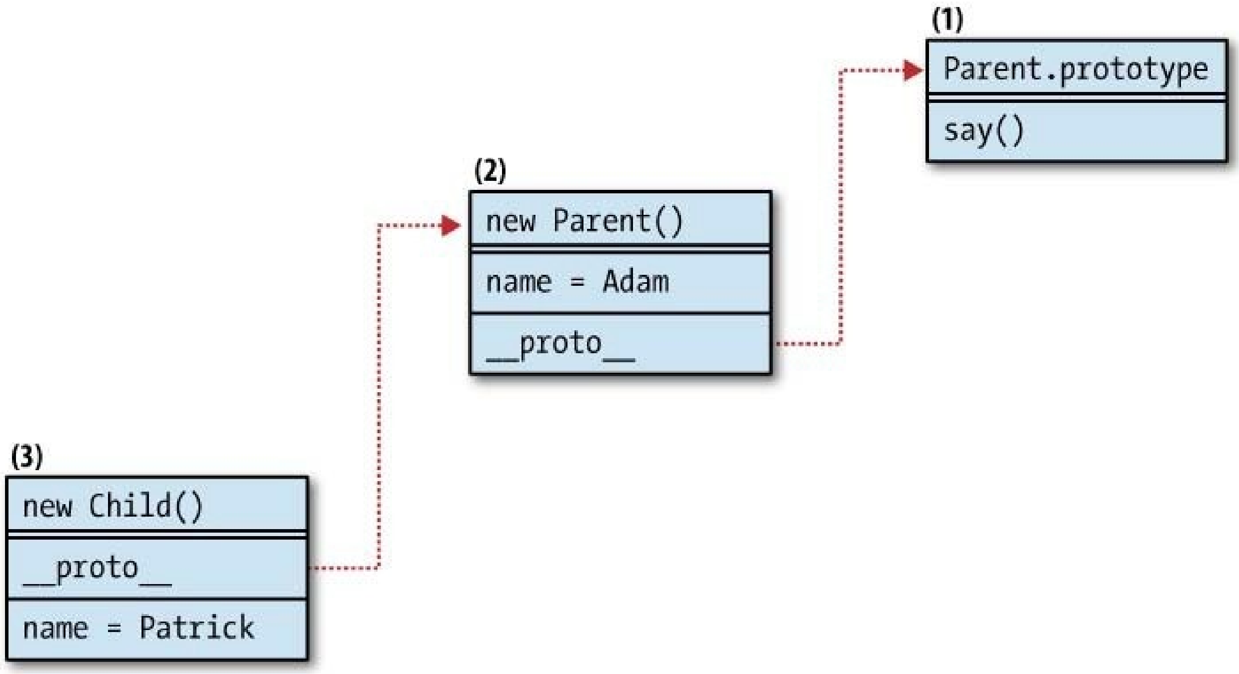


图6-3 继承并且给子对象添加属性后的原型链

设定kid.name并没有改变2号对象的name属性，但是它直接在3号对象上添加了自己的name属性。当kid.say()执行时，say方法在3号对象中找，然后是2号，最后到1号，像前面说的一样。但是这一次在找this.name（和kid.name一样）时很快，因为这个属性在3号对象中就被找到了。

如果通过delete kid.name的方式移除新添加的属性，那么2号对象的name属性将暴露出来并且在查找的时候被找到。

这种模式的缺点

这种模式的一个缺点是既继承了（父对象）“自己的属性”，也继承了原型中的属性。大部分情况下你可能并不需要“自己的属性”，因为它们更可能是为实例对象添加的，并不用于复用。

> 一个在构造函数上常用的规则是，用于复用的成员（译注：属性和方法）应该被添加到原型上。

在使用这个inherit()函数时另外一个不便是它不能够让你传参数给子构造函数，这些参数有可能是想再传给父构造函数的。考虑下面的例子：

```
var s = new Child('Seth');
s.say(); // "Adam"
```

这并不是我们期望的结果。事实上传递参数给父构造函数是可能的，但这样需要在每次需要一个子对象时再做一次继承，很不方便，因为需要不断地创建父对象。

类式继承2——借用构造函数

下面这种模式解决了从子对象传递参数到父对象的问题。它借用了父对象的构造函数，将子对象绑定到this，同时传入参数：

```
function Child(a, c, b, d) {
    Parent.apply(this, arguments);
}
```

使用这种模式时，只能继承在父对象的构造函数中添加到this的属性，不能继承原型上的成员。

使用借用构造函数的模式，子对象通过复制的方式继承父对象的成员，而不是像类式继承1中那样获得引用。下面的例子展示了这两者的不同：

```
//父构造函数
function Article() {
    this.tags = ['js', 'css'];
}
var article = new Article();

//BlogPost通过类式继承1（默认模式）从article继承
function BlogPost() {}
BlogPost.prototype = article;
var blog = new BlogPost();
//注意你不需要使用`new Article()`，因为已经有一个实例了

//StaticPage通过借用构造函数的方式从Article继承
function StaticPage() {
    Article.call(this);
}
var page = new StaticPage();

alert(article.hasOwnProperty('tags')); // true
alert(blog.hasOwnProperty('tags')); // false
alert(page.hasOwnProperty('tags')); // true
```

在上面的代码片段中，Article()被两种方式分别继承。默认模式使blog可以通过原型链访问到tags属性，所以它自己并没有tags属性，hasOwnProperty()返回false。page对象有自己的tags属性，因为它是使用借用构造函数的方式继承，复制（而不是引用）了tags属性。

注意在修改继承后的tags属性时的不同：

```
blog.tags.push('html');
page.tags.push('php');
alert(article.tags.join(', ')); // "js, css, html"
```

在这个例子中，blog对象修改了tags属性，同时，它也修改了父对象，因为实际上blog.tags和article.tags是引向同一个数组。而对pages.tags的修改并不影响父对象article，因为pages.tags在继承的时候是一份独立的拷贝。

原型链

我们来看一下当我们使用熟悉的Parent()和Child()构造函数和这种继承模式时原型链是什么样的。为了使用这种继承模式，Child()有明显变化：

```
//父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

//在原型上添加方法
Parent.prototype.say = function () {
    return this.name;
};

//子构造函数
function Child(name) {
    Parent.apply(this, arguments);
}

var kid = new Child("Patrick");
    kid.name; // "Patrick"
typeof kid.say; // "undefined"
```

如果看一下图6-4，就能发现new Child对象和Parent之间不再有链接。这是因为Child.prototype根本就没有被使用，它指向一个空对象。使用这种模式，kid拥有了自己的name属性，但是并没有继承say()方法，如果尝试调用它的话会出错。这种继承方式只是一种一次性地将父对象的属性复制为子对象的属性，并没有**proto**链接。

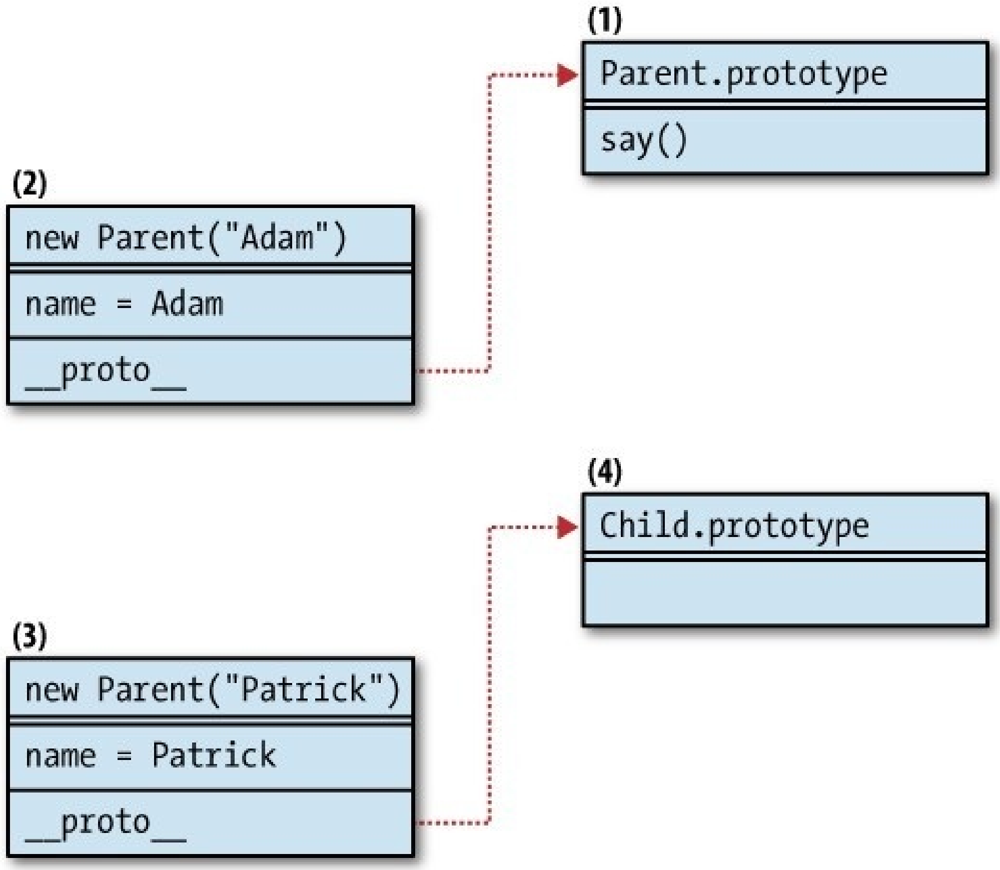


图6-4 使用借用构造函数模式时没有被关联的原型链

利用借用构造函数模式实现多继承

使用借用构造函数模式，可以通过借用多个构造函数的方式来实现多继承：

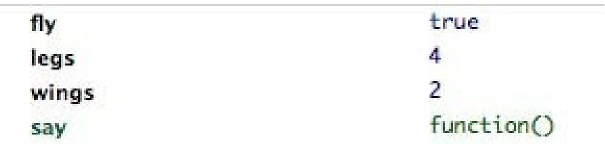
```
function Cat() {
    this.legs = 4;
    this.say = function () {
        return "meaowww";
    }
}

function Bird() {
    this.wings = 2;
    this.fly = true;
}

function CatWings() {
    Cat.apply(this);
    Bird.apply(this);
}

var jane = new CatWings();
console.dir(jane);
```

结果如图6-5，任何重复的属性都会以最后的一个值为准。



fly	true
legs	4
wings	2
say	function()

图6-5 在Firebug中查看CatWings对象

借用构造函数的利与弊

这种模式的一个明显的弊端就是无法继承原型。如前面所说，原型往往是添加可复用的方法和属性的地方，这样就不用在每个实例中再创建一遍。

这种模式的一个好处是获得了父对象自己成员的拷贝，不存在子对象意外改写父对象属性的风险。

那么，在上一个例子中，怎样使一个子对象也能够继承原型属性呢？怎样能使kid可以访问到say()方法呢？下一种继承模式解决了这个问题。

类式继承3——借用并设置原型

综合以上两种模式，首先借用父对象的构造函数，然后将子对象的原型设置为父对象的一个新实例：

```
function Child(a, c, b, d) {
    Parent.apply(this, arguments);
}
Child.prototype = new Parent();
```

这样做的好处是子对象获得了父对象自己的成员，也获得了父对象中可复用的（在原型中实现的）方法。子对象也可以传递任何参数给父构造函数。这种行为可能是最接近Java的，子对象继承了父对象的所有东西，同时可以安全地修改自己的属性而不用担心修改到父对象。

一个弊端是父构造函数被调用了两次，所以不是很高效。最后，（父对象）自己的属性（比如这个例子中的name）也被继承了两次。

我们来看一下代码并做一些测试：

```
//父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

//在原型上添加方法
Parent.prototype.say = function () {
    return this.name;
};

//子构造函数
function Child(name) {
    Parent.apply(this, arguments);
}
Child.prototype = new Parent();

var kid = new Child("Patrick");
kid.name; // "Patrick"
kid.say(); // "Patrick"
delete kid.name;
kid.say(); // "Adam"
```

跟前一种模式不一样，现在say()方法被正确地继承了。可以看到name也被继承了两次，在删除掉自己的拷贝后，在原型链上的另一个就被暴露出来了。

图6-6展示了这些对象之间的关系。这些关系有点像图6-3中展示的，但是获得这种关系的方法是不一样的。

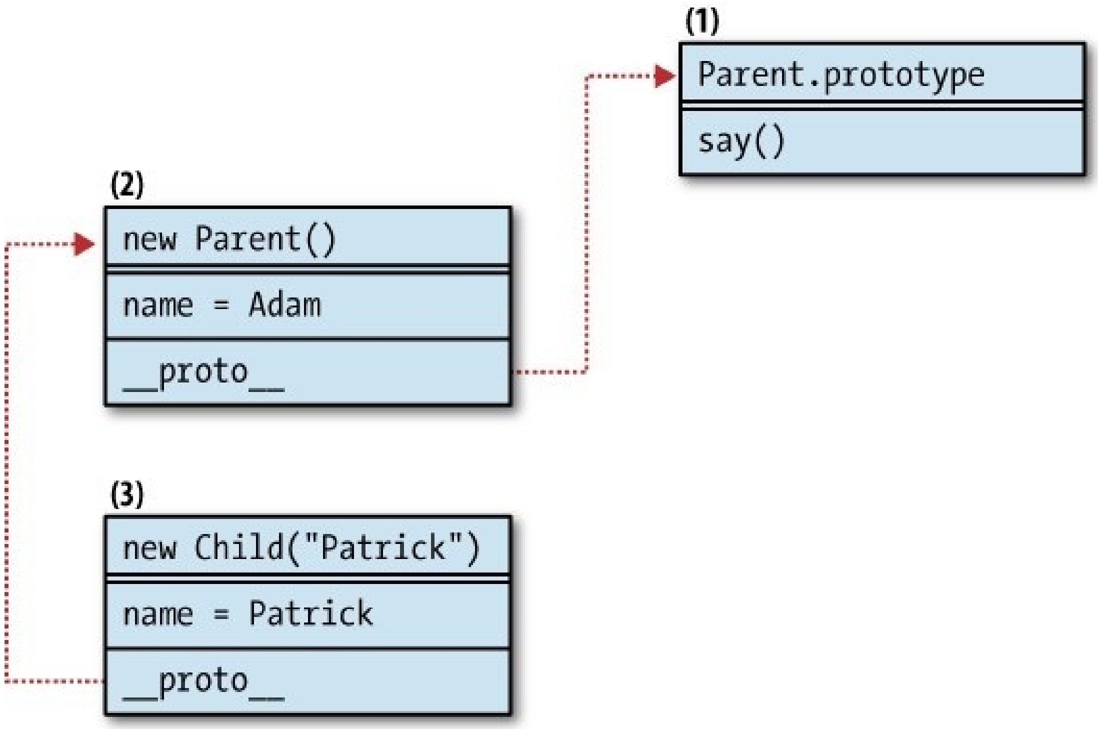


图6-6 除了继承“自己的属性”外，原型链也被保留了

类式继承4——共享原型

不像前一种类式继承模式需要调用两次父构造函数，下面这种模式根本不会涉及到调用父构造函数的问题。

一般的经验是将可复用的成员放入原型中而不是this。从继承的角度来看，则是任何应该被继承的成员都应该放入原型中。这样你只需要设定子对象的原型和父对象的原型一样即可：

```
function inherit(C, P) {
    C.prototype = P.prototype;
}
```

这种模式的原型链很短并且查找很快，因为所有的对象实际上共享着同一个原型。但是这样也有弊端，那就是如果子对象或者在继承关系中的某个地方的任何一个子对象修改这个原型，将影响所有的继承关系中的父对象。（译注：这里应该是指会影响到所有从这个原型中继承的对象。）

如图6-7，子对象和父对象共享同一个原型，都可以访问say()方法。但是，子对象不继承name属性。

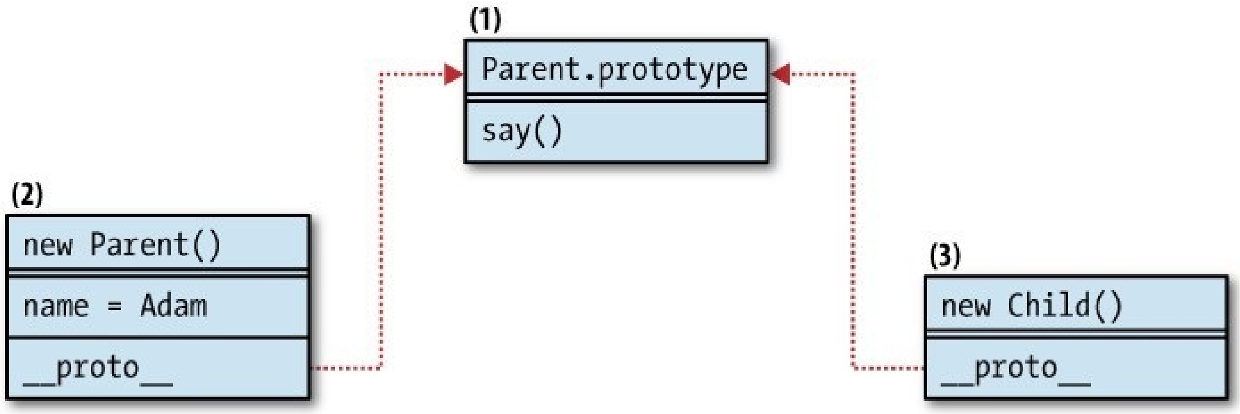


图6-7 （父子对象）共享原型时的关系

类式继承5——临时构造函数

下一种模式通过打断父对象和子对象原型的直接链接解决了共享原型时的问题，同时还从原型链中获得其它的好处。

下面是这种模式的一种实现方式，F()函数是一个空函数，它充当了子对象和父对象的代理。F()的prototype属性指向父对象的原型。子对象的原型是一这个空函数的一个实例：

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
}
```

这种模式有一种和默认模式（类式继承1）明显不一样的行为，因为在这里子对象只继承原型中的属性（图6-8）。

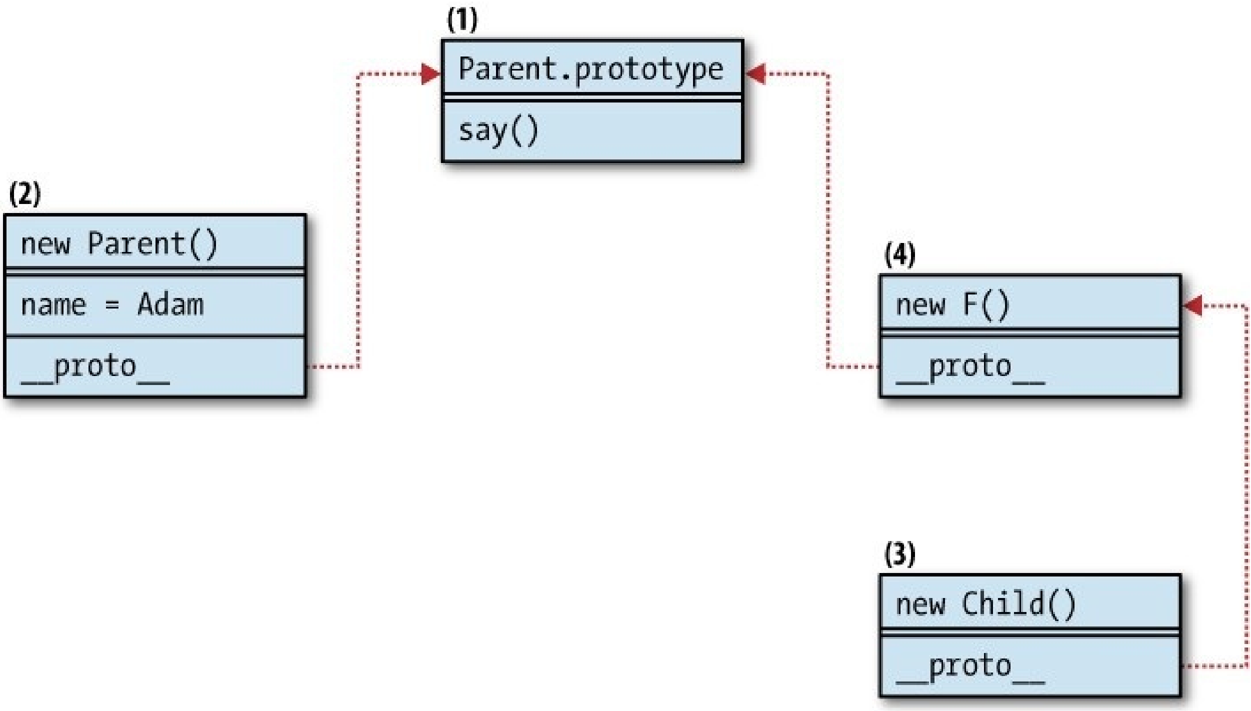


图6-8 使用临时（代理）构造函数F()实现类式继承

这种模式通常情况下都是一种很棒的选择，因为原型本来就是存放复用成员的地方。在这种模式中，父构造函数添加到this中的任何成员都不会被继承。

我们来创建一个子对象并且检查一下它的行为：

```
var kid = new Child();
```

如果你访问kid.name将得到undefined。在这个例子中，name是父对象自己的属性，而在继承的过程中我们并没有调用new Parent()，所以这个属性并没有被创建。当访问kid.say()时，它在3号对象中不可用，所以在原型链中查找，4号对象也没有，但是1号对象有，它在内在中的位置会被所有从Parent()创建的构造函数和子对象所共享。

存储父类（Superclass）

在上一种模式的基础上，还可以添加一个指向原始父对象的引用。这很像其它语言中访问超类（superclass）的情况，有时候很方便。

我们将这个属性命名为“uber”，因为“super”是一个保留字，而“superclass”则可能误导别人认为JavaScript拥有类。下面是这种类式继承模式的一个改进版实现：

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
    C.uber = P.prototype;
}
```

重置构造函数引用

这个近乎完美的模式上还需要做的最后一件事情就是重置构造函数（constructor）的指向，以便未来在某个时刻能被正确地使用。

如果不重置构造函数的指向，那所有的子对象都会认为Parent()是它们的构造函数，而这个结果完全没有用。使用前面的inherit()的实现，你可以观察到这种行为：

```
// parent, child, inheritance
function Parent() {}
function Child() {}
inherit(Child, Parent);

// testing the waters
var kid = new Child();
kid.constructor.name; // "Parent"
kid.constructor === Parent; // true
```

constructor属性很少用，但是在运行时检查对象很方便。你可以重新将它指向期望的构造函数而不影响功能，因为这个属性更多是“信息性”的。（译注：即它更多的时候是在提供信息而不是参与到函数功能中。）

最终，这种类式继承的Holy Grail版本看起来是这样的：

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
    C.uber = P.prototype;
    C.prototype.constructor = C;
}
```

类似这样的函数也存在于YUI库（也许还有其它库）中，它将类式继承的方法带给了没有类的语言。如果你决定使用类式继承，那么这是最好的方法。

> “代理函数”或者“代理构造函数”也是指这种模式，因为临时构造函数是被用作获取父构造函数原型的代理。

一种常见的对Holy Grail模式的优化是避免每次需要继承的时候都创建一个临时（代理）构造函数。事实上创建一次就足够了，以后只需要修改它的原型即可。你可以用一个立即执行的函数来将代理函数存储到闭包中：

```
var inherit = (function () {
    var F = function () {};
    return function (C, P) {
        F.prototype = P.prototype;
        C.prototype = new F();
        C.uber = P.prototype;
        C.prototype.constructor = C;
    }
}());
```

Klass

有很多JavaScript类库模拟了类，创造了新的语法糖。具体的实现方式可能会不一样，但是基本上都有一些共性，包括：

- 有一个约定好名字的方法，如initialize、_init或者其它相似的名字，会被自动调用，来充当类的构造函数。
- 类可以从其它类继承
- 在子类中可以访问到父类（superclass）

> 我们在这里做一下变化，在本章的这部分自由地使用“class”单词，因为主题就是模拟类。

为避免讨论太多细节，我们来看一下JavaScript中一种模拟类的实现。首先，这种解决方案从客户的角度来看将如何被使用？

```
var Man = klass(null, {
    __construct: function (what) {
        console.log("Man's constructor");
        this.name = what;
    },
    getName: function () {
        return this.name;
    }
});
```

这种语法糖的形式是一个名为klass()的函数。在一些实现方式中，它可能是Klass()构造函数或者是增强的Object.prototype，但是在这个例子中，我们让它只是一个简单的函数。

这个函数接受两个参数：一个被继承的类和通过对象字面量提供的新类的实现。受PHP的影响，我们约定类的构造函数必须是一个名为__construct的方法。在前面的代码片段中，建立了一个名为Man的新类，并且它不继承任何类（意味着继承自Object）。Man类有一个在__construct建立的自己的属性name和一个方法getName()。这个类是一个构造函数，所以下面的代码将正常工作（并且看起来像类实例化的过程）：

```
var first = new Man('Adam'); // logs "Man's constructor"
first.getName(); // "Adam"
```

现在我们来扩展这个类，创建一个SuperMan类：

```
var SuperMan = klass(Man, {
    __construct: function (what) {
        console.log("SuperMan's constructor");
    },
    getName: function () {
        var name = SuperMan.uber.getName.call(this);
        return "I am " + name;
    }
});
```

这里，klass()的第一个参数是将被继承的Man类。值得注意的是，在getName()中，父类的getName()方法首先通过SuperMan类的uber静态属性被调用。我们来测试一下：

```
var clark = new SuperMan('Clark Kent');
clark.getName(); // "I am Clark Kent"
```

第一行在console中记录了“Man's constructor”，然后是“Superman's constructor”。在一些语言中，父类的构造函数在子类构造函数被调用的时候会自动执行，这个特性也可以模拟。

用instanceof运算符测试返回希望的结果：

```
clark instanceof Man; // true
clark instanceof SuperMan; // true
```

最后，我们来看一下klass()函数是怎样实现的：

```
var klass = function (Parent, props) {

    var Child, F, i;

    // 1.
    // new constructor
    Child = function () {
        if (Child.uber && Child.uber.hasOwnProperty("__construct")) {
            Child.uber.__construct.apply(this, arguments);
        }
        if (Child.prototype.hasOwnProperty("__construct")) {
            Child.prototype.__construct.apply(this, arguments);
        }
    };

    // 2.
    // inherit
    Parent = Parent || Object;
    F = function () {};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.uber = Parent.prototype;
    Child.prototype.constructor = Child;

    // 3.
    // add implementation methods
    for (i in props) {
        if (props.hasOwnProperty(i)) {
            Child.prototype[i] = props[i];
        }
    }

    // return the "class"
    return Child;
};
```

这个klass()实现有三个明显的部分：

1. 创建Child()构造函数，这也是最后返回的将被作为类使用的函数。在这个函数里面，如果__construct方法存在的话将被调用。同样是在父类的__construct（如果存在）被调用前使用静态的uber属性。也可能存在uber没有定义的情况——比如从Object继承，因为它

是在Man类中被定义的。

2. 第二部分主要完成继承。只是简单地使用前面章节讨论过的Holy Grail类式继承模式。只有一个东西是新的：如果Parent没有传值的话，设定Parent为Object。
3. 最后一部分是类真正定义的地方，循环需要实现的方法（如例子中的__constructt和getName），并将它们添加到Child的原型中。

什么时候使用这种模式？其实，最好是能避免则避免，因为它带来了在这门语言中不存在的完整的类的概念，会让人疑惑。使用它需要学习新的语法和新的规则。也就是说，如果你或者你的团队对类感到习惯并且同时对原型感到不习惯，这种模式可能是一个可以探索的方向。这种模式允许你完全忘掉原型，好处就是你可以将语法变种得像其它你所喜欢的语言一样。

原型继承

现在，让我们从一个叫作“原型继承”的模式来讨论没有类的现代继承模式。在这种模式中，没有任何类进来，在这里，一个对象继承自另外一个对象。你可以这样理解它：你有一个想复用的对象，然后你想创建第二个对象，并且获得第一个对象的功能。下面是这种模式的用法：

```
//需要继承的对象
var parent = {
    name: "Papa"
};

//新对象
var child = object(parent);

//测试
alert(child.name); // "Papa"
```

在这个代码片段中，有一个已经存在的使用对象字面量创建的对象叫parent，我们想创建一个和parent有相同的属性和方法的对象叫child。child对象使用object()函数创建。这个函数在JavaScript中并不存在（不要与构造函数Object()混淆），所以我们来看看怎样定义它。

与Holy Grail类式继承相似，可以使用一个空的临时构造函数F()，然后设定F()的原型为parent对象。最后，返回一个临时构造函数的新实例。

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

图6-9展示了使用原型继承时的原型链。在这里child总是以一个空对象开始，它没有自己的属性但通过原型链（__proto__）拥有父对象的所有功能。

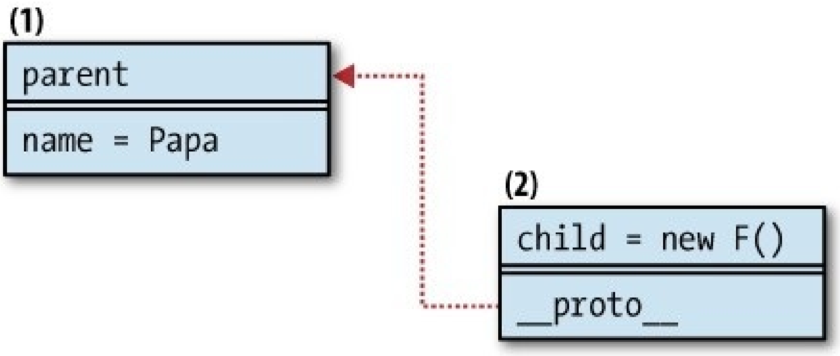


图6-9 原型继承模式

讨论

在原型继承模式中，parent不需要使用对象字面量来创建。（尽管这是一种更觉的方式。）可以使用构造函数来创建parent。注意，如果你这样做，那么自己的属性和原型上的属性都将被继承：

```
// parent constructor
function Person() {
    // an "own" property
    this.name = "Adam";
}
// a property added to the prototype
Person.prototype.getName = function () {
    return this.name;
};

// create a new person
var papa = new Person();
// inherit
var kid = object(papa);

// test that both the own property
// and the prototype property were inherited
kid.getName(); // "Adam"
```

在这种模式的另一个变种中，你可以选择只继承已存在的构造函数的原型对象。记住，对象继承自对象，不管父对象是怎么创建的。这是前面例子的一个修改版本：

```
// parent constructor
function Person() {
    // an "own" property
    this.name = "Adam";
}
// a property added to the prototype
Person.prototype.getName = function () {

};

// inherit
var kid = object(Person.prototype);

typeof kid.getName; // "function", because it was in the prototype
typeof kid.name; // "undefined", because only the prototype was inherited
```

例外的ECMAScript 5

在ECMAScript 5中，原型继承已经正式成为语言的一部分。这种模式使用Object.create方法来实现。换句话说，你不再需要自己去写类似object()的函数，它是语言原生的了：

```
var child = Object.create(parent);
```

Object.create()接收一个额外的参数——一个对象。这个额外对象中的属性将被作为自己的属性添加到返回的子对象中。这让我们可以很方便地将继承和创建子对象在一个方法调用中实现。例如：

```
var child = Object.create(parent, {
    age: { value: 2 } // ECMA5 descriptor
});
child.hasOwnProperty("age"); // true
```

你可能也会发现原型继承模式已经在一些JavaScript类库中实现了，比如，在YUI3中，它是Y.Object()方法：

```
YUI().use('*', function (Y) {
    var child = Y.Object(parent);
});
```

通过复制属性继承

让我们来看一下另外一种继承模式——通过复制属性继承。在这种模式中，一个对象通过简单地复制另一个对象来获得功能。下面是一个简单的实现这种功能的extend()函数：

```
function extend(parent, child) {
    var i;
    child = child || {};
    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            child[i] = parent[i];
        }
    }
        return child;
}
```

这是一个简单的实现，仅仅是遍历了父对象的成员然后复制它们。在这个实现中，child是可选参数，如果它没有被传入一个已有的对象，那么一个全新的对象将被创建并被返回：

```
var dad = {name: "Adam"};
var kid = extend(dad);
kid.name; // "Adam"
```

上面给出的实现叫作对象的“浅拷贝”（shallow copy）。另一方面，“深拷贝”是指检查准备复制的属性本身是否是对象或者数组，如果是，也遍历它们的属性并复制。如果使用浅拷贝的话（因为在JavaScript中对象是按引用传递），如果你改变子对象的一个属性，而这个属性恰好是一个对象，那么你也会改变父对象。实际上这对方法来说可能很好（因为函数也是对象，也是按引用传递），但是当遇到其它的对象和数组的时候可能会有些意外情况。考虑这种情况：

```
var dad = {
    counts: [1, 2, 3],
    reads: {paper: true}
};
var kid = extend(dad);
kid.counts.push(4);
dad.counts.toString(); // "1,2,3,4"
dad.reads === kid.reads; // true
```

现在让我们来修改一下extend()函数以便做深拷贝。所有你需要做的事情只是检查一个属性的类型是否是对象，如果是，则递归遍历它的属性。另外一个需要做的检查是这个对象是真的对象还是数组。我们可以使用第3章讨论过的数组检查方式。最终深拷贝版的extend()是这样的：

```
function extendDeep(parent, child) {
    var i,
        toStr = Object.prototype.toString,
        astr = "[object Array]";

    child = child || {};

    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            if (typeof parent[i] === "object") {
                child[i] = (toStr.call(parent[i]) === astr) ? [] : {};
                extendDeep(parent[i], child[i]);
            } else {
                child[i] = parent[i];
            }
        }
    }
    return child;
}
```

现在测试时这个新的实现给了我们对象的真实拷贝，所以子对象不会修改父对象：

```
var dad = {
    counts: [1, 2, 3],
    reads: {paper: true}
};
var kid = extendDeep(dad);

kid.counts.push(4);
kid.counts.toString(); // "1,2,3,4"
dad.counts.toString(); // "1,2,3"

dad.reads === kid.reads; // false
kid.reads.paper = false;
kid.reads.web = true;
dad.reads.paper; // true
```

通过复制属性继承的模式很简单且应用很广泛。例如Firebug（JavaScript写的Firefox扩展）有一个方法叫extend()做浅拷贝，jQuery的extend()方法做深拷贝。YUI3提供了一个叫作Y.clone()的方法，它创建一个深拷贝并且通过绑定到子对象的方式复制函数。（本章后面将有更多关于绑定的内容。）

这种模式并不高深，因为根本没有原型牵涉进来，而只跟对象和它们的属性有关。

混元（Mix-ins）

既然谈到了通过复制属性来继承，就让我们顺便多说一点，来讨论一下“混元”模式。除了前面说的从一个对象复制，你还可以从任意多数量的对象中复制属性，然后将它们混在一起组成一个新对象。

实现很简单，只需要遍历传入的每个参数然后复制它们的每个属性：

```
function mix() {
    var arg, prop, child = {};
    for (arg = 0; arg < arguments.length; arg += 1) {
        for (prop in arguments[arg]) {
            if (arguments[arg].hasOwnProperty(prop)) {
                child[prop] = arguments[arg][prop];
            }
        }
    }
    return child;
}
```

现在我们有了一个通用的混元函数，我们可以传递任意数量的对象进去，返回的结果将是一个包含所有传入对象属性的新对象。下面是用法示例：

```
var cake = mix(
    {eggs: 2, large: true},
    {butter: 1, salted: true},
    {flour: "3 cups"},
    {sugar: "sure!"}
);
```

图6-10展示了在Firebug的控制台中用console.dir(cake)展示出来的混元后cake对象的属性。

butter	1
eggs	2
flour	"3 cups"
large	true
salted	true
sugar	"sure!"

图6-10 在Firebug中查看cake对象

> 如果你习惯了某些将混元作为原生部分的语言，那么你可能期望修改一个或多个父对象时也影响子对象。但在这个实现中这是不会发生的事情。这里我们只是简单地遍历、复制自己的属性，并没有与父对象的链接。

借用方法

有时候会有这样的情况：你希望使用某个已存在的对象的一两个方法，你希望能复用它们，但是又真的不希望和那个对象产生继承关系，因为你只希望使用你需要的那一两个方法，而不继承那些你永远用不到的方法。受益于函数方法call()和apply()，通过借用方法模式，这是可行的。在本书中，你其实已经见过这种模式了，甚至在本章extendDeep()的实现中也有用到。

如你所熟知的一样，在JavaScript中函数也是对象，它们有一些有趣的方法，比如call()和apply()。这两个方法的唯一区别是后者接受一个参数数组以传入正在调用的方法，而前者只接受一个一个的参数。你可以使用这两个方法来从已有的对象中借用方法：

```
//call() example
notmyobj.doStuff.call(myobj, param1, p2, p3);
// apply() example
notmyobj.doStuff.apply(myobj, [param1, p2, p3]);
```

在这个例子中有一个对象myobj，而且notmyobj有一个用得着的方法叫doStuff()。你可以简单地临时借用doStuff()方法，而不用处理继承然后得到一堆myobj中你永远不会用的方法。

你传一个对象和任意的参数，这个被借用的方法会将this绑定到你自己的对象上。简单地说，你的对象会临时假装成另一个对象以使用它的方法。这就像实际上获得了继承但又免除了“继承税”（指你不需要的属性和方法）。

例：从数组借用

这种模式的一种常见用法是从数组借用方法。

数组有很多很有用但是一些“类数组”对象（如arguments）不具备的方法。所以arguments可以借用数组的方法，比如slice()。这是一个例子：

```
function f() {
    var args = [].slice.call(arguments, 1, 3);
        return args;
    }

// example
f(1, 2, 3, 4, 5, 6); // returns [2,3]
```

在这个例子中，有一个空数组被创建了，因为要借用它的方法。同样的事情也可以使用一种看起来代码更长的方法来做，那就是直接从数组的原型中借用方法，使用Array.prototype.slice.call(...)。这种方法代码更长一些，但是不用创建一个空数组。

借用并绑定

当借用方法的时候，不管是通过call()/apply()还是通过简单的赋值，方法中的this指向的对象都是基于调用的表达式来决定的。但是有时候最好的使用方式是将this的值锁定或者提前绑定到一个指定的对象上。

我们来看一个例子。这是一个对象one，它有一个say()方法：

```
var one = {
    name: "object",
    say: function (greet) {
        return greet + ", " + this.name;
    }
};

// test
one.say('hi'); // "hi, object"
```

现在另一个对象two没有say()方法，但是它可以从one借用：

```
var two = {
    name: "another object"
};

one.say.apply(two, ['hello']); // "hello, another object"
```

在这个例子中，say()方法中的this指向了two，this.name是“another object”。但是如果在某些场景下你将th函数赋值给了全局变量或者是将这个函数作为回调，会发生什么？在客户端编程中有非常多的事件和回调，所以这种情况经常发生：

```
// assigning to a variable
// `this` will point to the global object
var say = one.say;
say('hoho'); // "hoho, undefined"

// passing as a callback
var yetanother = {
    name: "Yet another object",
    method: function (callback) {
        return callback('Hola');
    }
};
yetanother.method(one.say); // "Holla, undefined"
```

在这两种情况中say()中的this都指向了全局对象，所以代码并不像我们想象的那样正常工作。要修复（换言之，绑定）一个方法的对象，我们可以用一个简单的函数，像这样：

```
function bind(o, m) {
    return function () {
        return m.apply(o, [].slice.call(arguments));
    };
}
```

这个bind()函数接受一个对象o和一个方法m，然后把它们绑定在一起，再返回另一个函数。返回的函数通过闭包可以访问到o和m。也就是说，即使在bind()返回之后，内层的函数仍然可以访问到o和m，而o和m会始终指向原始的对象和方法。让我们用bind()来创建一个新函数：

```
var twosay = bind(two, one.say);
twosay('yo'); // "yo, another object"
```

正如你看到的，尽管twosay()是作为一个全局函数被创建的，但this并没有指向全局对象，而是指向了通过bind()传入的对象two。不论无何调用twosay()，this将始终指向two。

绑定是奢侈的，你需要付出的代价是一个额外的闭包。

Function.prototype.bind()

ECMAScript5在Function.prototype中添加了一个方法叫bind()，使用时和apply和call()一样简单。所以你可以这样写：

```
var newFunc = obj.someFunc.bind(myobj, 1, 2, 3);
```

这意味着将someFunc()主myobj绑定了并且传入了someFunc()的前三个参数。这也是一个在第4章讨论过的部分应用的例子。

让我们来看一下当你的程序跑在低于ES5的环境中时如何实现Function.prototype.bind()：

```
if (typeof Function.prototype.bind === "undefined") {
    Function.prototype.bind = function (thisArg) {
        var fn = this,
        slice = Array.prototype.slice,
        args = slice.call(arguments, 1);

        return function () {
            return fn.apply(thisArg, args.concat(slice.call(arguments)));
        };
    };
}
```

这个实现可能看起来有点熟悉，它使用了部分应用，将传入bind()的参数串起来（除了第一个参数），然后在被调用时传给bind()返回的新函数。这是用法示例：

```
var twosay2 = one.say.bind(two);
twosay2('Bonjour'); // "Bonjour, another object"
```

在这个例子中，除了绑定的对象外，我们没有传任何参数给bind()。下一个例子中，我们来传一个用于部分应用的参数：

```
var twosay3 = one.say.bind(two, 'Enchanté');
twosay3(); // "Enchanté, another object"
```

小结

在JavaScript中，继承有很多种方案可以选择。学习和理解不同的模式是有好处的，因为这可以增强你对这门语言的掌握能力。在本章中你看到了很多类式继承和现代继承的方案。

但是，也许在开发过程中继承并不是你经常面对的一个问题。这一部分是因为这个问题已经被使用某种方式或者某个你使用的类库解决了，另一部分是因为你不需要在JavaScript中建立很长很复杂的继承链。在静态强类型语言中，继承可能是唯一可以利用代码的方法，但在JavaScript中你可能有更多更简单更优化的方法，包括借用方法、绑定、复制属性、混元等。

记住，代码复用才是目标，继承只是达成这个目标的一种手段。

设计模式

在GoF（Gang of Four）的书中提出的设计模式为面向对象的软件设计中遇到的一些普遍问题提供了解决方案。它们已经诞生很久了，而且被证实在很多情况下是很有效的。这正是你需要熟悉它的原因，也是我们要讨论它的原因。

尽管这些设计模式跟语言和具体的实现方式无关，但它们多年来被关注到的方面仍然主要是在强类型静态语言比如C++和Java中的应用。

JavaScript作为一种基于原型的弱类型动态语言，使得有些时候实现某些模式时相当简单，甚至不费吹灰之力。

让我们从第一个例子——单例模式——来看一下在JavaScript中和静态的基于类的语言有什么不同。

单例

单例模式的核心思想是让指定的类只存在唯一一个实例。这意味着当你第二次使用相同的类去创建对象的时候，你得到的应该和第一次创建的是同一个对象。

这如何应用到JavaScript中呢？在JavaScript中没有类，只有对象。当你创建一个对象时，事实上根本没有另一个对象和它一样，这个对象其实已经是一个单例。使用对象字面量创建一个简单的对象也是一种单例的例子：

```
var obj = {
    myprop: 'my value'
};
```

在JavaScript中，对象永远不会相等，除非它们是同一个对象，所以即使你创建一个看起来完全一样的对象，它也不会和前面的对象相等：

```
var obj2 = {
    myprop: 'my value'
};
obj === obj2; // false
obj == obj2; // false
```

所以你可以说当你每次使用对象字面量创建一个对象的时候就是在创建一个单例，并没有特别的语法迁涉进来。

> 需要注意的是，有的时候当人们在JavaScript中提出“单例”的时候，它们可能是在指第5章讨论过的“模块模式”。

使用new

JavaScript没有类，所以一字一句地说单例的定义并没有什么意义。但是JavaScript有使用new、通过构造函数来创建对象的语法，有时候你可能需要这种语法下的一个单例实现。这也就是说当你使用new、通过同一个构造函数来创建多个对象的时候，你应该只是得到同一个对象的不同引用。

> 温馨提示：从一个实用模式的角度来说，下面的讨论并不是那么有用，只是更多地在实践模拟一些语言中关于这个模式的一些问题的解决方案。这些语言主要是（静态强类型的）基于类的语言，在这些语言中，函数并不是“一等公民”。

下面的代码片段展示了期望的结果（假设你忽略了多元宇宙的设想，接受了只有一个宇宙的观点）：

```
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

在这个例子中，uni只在构造函数第一次被调用时创建。第二次（以及后续更多次）调用时，同一个uni对象被返回。这就是为什么uni === uni2的原因——因为它们实际上是同一个对象的两个引用。那么怎么在JavaScript达到这个效果呢？

当对象实例this被创建时，你需要在Universe构造函数中缓存它，以便在第二次调用的时候返回。有几种选择可以达到这种效果：

- 你可以使用一个全局变量来存储实例。不推荐使用这种方法，因为通常我们认为使用全局变量是不好的。而且，任何人都可以改写全局变量的值，甚至可能是无意中改写。所以我们不再讨论这种方案。
- 你也可以将对象实例缓存在构造函数的属性中。在JavaScript中，函数也是对象，所以它们也可以有属性。你可以写一些类似Universe.instance的属性来缓存对象。这是一种漂亮干净的解决方案，不足之处是instance属性仍然是可以被公开访问的，别人写的代码可能修改它，这样就会失去这个实例。
- 你可以将实例包裹在闭包中。这可以保持实例是私有的，不会在构造函数之外被修改，代价是一个额外的闭包。

让我们来看一下第二种和第三种方案的实现示例。

将实例放到静态属性中

下面是一个将唯一的实例放入Universe构造函数的一个静态属性中的例子：

```
function Universe() {

// do we have an existing instance?
if (typeof Universe.instance === "object") {
    return Universe.instance;
}

// proceed as normal
this.start_time = 0;
this.bang = "Big";

// cache
Universe.instance = this;

// implicit return:
// return this;
}

// testing
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

如你所见，这是一种直接有效的解决方案，唯一的缺陷是instance是可被公开访问的。一般来说它被其它代码误删改的可能是很小的（起码比全局变量instance要小得多），但是仍然是有可能的。

将实例放到闭包中

另一种实现基于类的单例模式的方法是使用一个闭包来保护这个唯一的实例。你可以通过第5章讨论过的“私有静态成员模式”来实现。唯一的秘密就是重写构造函数：

```
function Universe() {

    // the cached instance
    var instance = this;

    // proceed as normal
    this.start_time = 0;
    this.bang = "Big";

    // rewrite the constructor
    Universe = function () {
        return instance;
    };
}

// testing
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

第一次调用时，原始的构造函数被调用并且正常返回this。在后续的调用中，被重写的构造函数被调用。被重写怕这个构造函数可以通过闭包访问私有的instance变量并且将它返回。

这个实现实际上也是第4章讨论的自定义函数的又一个例子。如我们讨论过的一样，这种模式的缺点是被重写的函数（在这个例子中就是构造函数Universe()）将丢失那些在初始定义和重新定义之间添加的属性。在这个例子中，任何添加到Universe()的原型上的属性将不会被链接

到使用原来的实现创建的实例上。（注：这里的“原来的实现”是指实例是由未被重写的构造函数创建的，而Universe()则是被重写的构造函数。）

下面我们通过一些测试来展示这个问题：

```
// adding to the prototype
Universe.prototype.nothing = true;

var uni = new Universe();

// again adding to the prototype
// after the initial object is created
Universe.prototype.everything = true;

var uni2 = new Universe();

Testing:
// only the original prototype was
// linked to the objects
uni.nothing; // true
uni2.nothing; // true
uni.everything; // undefined
uni2.everything; // undefined

// that sounds right:
uni.constructor.name; // "Universe"

// but that's odd:
uni.constructor === Universe; // false
```

uni.constructor不再和Universe()相同的原因是uni.constructor仍然是指向原来的构造函数，而不是被重新定义的那个。

如果一定被要求让prototype和constructor的指向像我们期望的那样，可以通过一些调整来做到：

```
function Universe() {

    // the cached instance
    var instance;

    // rewrite the constructor
    Universe = function Universe() {
        return instance;
    };

    // carry over the prototype properties
    Universe.prototype = this;

    // the instance
    instance = new Universe();

    // reset the constructor pointer
    instance.constructor = Universe;

    // all the functionality
    instance.start_time = 0;
    instance.bang = "Big";

    return instance;
}
```

现在所有的测试结果都可以像我们期望的那样了：

```
// update prototype and create instance
Universe.prototype.nothing = true; // true
var uni = new Universe();
Universe.prototype.everything = true; // true
var uni2 = new Universe();

// it's the same single instance
uni === uni2; // true

// all prototype properties work
// no matter when they were defined
uni.nothing && uni.everything && uni2.nothing && uni2.everything; // true
// the normal properties work
uni.bang; // "Big"
// the constructor points correctly
uni.constructor === Universe; // true
```

另一种可选的解决方案是将构造函数和实例包在一个立即执行的函数中。当构造函数第一次被调用的时候，它返回一个对象并且将私有的instance指向它。在后续调用时，构造函数只是简单地返回这个私有变量。在这种新的实现下，前面所有的测试代码也会和期望的一样：

```
var Universe;

(function () {

    var instance;

    Universe = function Universe() {

    if (instance) {
        return instance;
    }

    instance = this;

    // all the functionality
    this.start_time = 0;
    this.bang = "Big";

    };

}());
```

工厂模式

使用工厂模式的目的就是创建对象。它通常被在类或者类的静态方法中实现，目的是：

- 执行在建立相似的对象时进行的一些重复操作
- 让工厂的使用者在编译阶段创建对象时不必知道它的特定类型（类）

第二点在静态的基于类的语言中更重要，因为在（编译阶段）提前不知道类的情况下，创建类的实例是不普通的行为。但在JavaScript中，这部分的实现却是相当容易的事情。

使用工厂方法（或类）创建的对象被设计为从同一个父对象继承；它们是特定的实现一些特殊功能的子类。有些时候这个共同的父对象就是包含工厂方法的同一个类。

我们来看一个示例实现，我们有：

- 一个共同的父构造函数CarMaker。
- CarMaker的一个静态方法叫factory()，用来创建car对象。
- 特定的从CarMaker继承而来的构造函数CarMaker.Compact，CarMaker.SUV，CarMaker.Convertible。它们都被定义为父构造函数的静态属性以便保持全局空间干净，同时在需要的时候我们也知道在哪里找到它们。

我们来看一下已经完成的实现会怎么被使用：

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');
corolla.drive(); // "Vroom, I have 4 doors"
solstice.drive(); // "Vroom, I have 2 doors"
cherokee.drive(); // "Vroom, I have 17 doors"
```

这一段：

```
var corolla = CarMaker.factory('Compact');
```

可能是工厂模式中最知名的。你有一个方法可以在运行时接受一个表示类型的字符串，然后它创建并返回了一个和请求的类型一样的对象。这里没有使用new的构造函数，也没有看到任何对象字面量，仅仅只有一个函数根据一个字符串指定的类型创建了对象。

这里是一个工厂模式的示例实现，它能让上面的代码片段工作：

```
// parent constructor
function CarMaker() {}

// a method of the parent
CarMaker.prototype.drive = function () {
    return "Vroom, I have " + this.doors + " doors";
};

// the static factory method
CarMaker.factory = function (type) {
    var constr = type,
    newcar;

    // error if the constructor doesn't exist
    if (typeof CarMaker[constr] !== "function") {
        throw {
            name: "Error",
            message: constr + " doesn't exist"
        };
    }

    // at this point the constructor is known to exist
    // let's have it inherit the parent but only once
    if (typeof CarMaker[constr].prototype.drive !== "function") {
        CarMaker[constr].prototype = new CarMaker();
    }
    // create a new instance
    newcar = new CarMaker[constr]();
    // optionally call some methods and then return...
    return newcar;
};

// define specific car makers
CarMaker.Compact = function () {
    this.doors = 4;
};
CarMaker.Convertible = function () {
    this.doors = 2;
};
CarMaker.SUV = function () {
    this.doors = 24;
};
```

工厂模式的实现中没有什么是特别困难的。你需要做的仅仅是寻找请求类型的对象的构造函数。在这个例子中，使用了一个简单的名字转换以便映射对象类型和创建对象的构造函数。继承的部分只是一个公共的重复代码片段的示例，它可以被放到工厂方法中而不是被每个构造函数的类型重复。（译注：指通过原型继承的代码可以在factory方法以外执行，而不是放到factory中每调用一次都要执行一次。）

内置对象工厂

作为一个“野生的工厂”的例子，我们来看一下内置的全局构造函数Object()。它的行为很像工厂，因为它根据不同的输入创建不同的对象。如果传入一个数字，它会使用Number()构造函数创建一个对象。在传入字符串和布尔值的时候也会发生同样的事情。任何其它的值（包括空值）将会创建一个正常的对象。

下面是这种行为的例子和测试，注意Object调用时可以不用加new：

```
var o = new Object(),
    n = new Object(1),
    s = Object('1'),
    b = Object(true);

// test
o.constructor === Object; // true
n.constructor === Number; // true
s.constructor === String; // true
b.constructor === Boolean; // true
```

Object()也是一个工厂这一事实可能没有太多实际用处，仅仅是觉得值得作为一个例子提一下，告诉我们工厂模式是随处可见的。

迭代器

在迭代器模式中，你有一些含有有序聚合数据的对象。这些数据可能在内部用一种复杂的结构存储着，但是你希望提供一种简单的方法来访问这种结构中的每个元素。数据的使用者不需要知道你是怎样组织你的数据的，他们只需要操作一个个独立的元素。

在迭代器模式中，你的对象需要提供一个next()方法。按顺序调用next()方法必须返回序列中的下一个元素，但是“下一个”在你的特定的数据结构中指什么是由你自己来决定的。

假设你的对象叫agg，你可以通过简单地在循环中调用next()来访问每个数据元素，像这样：

```
var element;
while (element = agg.next()) {
    // do something with the element ...
    console.log(element);
}
```

在迭代器模式中，聚合对象通常也会提供一个方便的方法hasNext()，这样对象的使用者就可以知道他们已经获取到你数据的最后一个元素。当使用另一种方法——hasNext()——来按顺序访问所有元素时，是像这样的：

```
while (agg.hasNext()) {
    // do something with the next element...
    console.log(agg.next());
}
```

装饰器

在装饰器模式中，一些额外的功能可以在运行时被动态地添加到一个对象中。在静态的基于类的语言中，处理这个问题可能是个挑战，但是在JavaScript中，对象本来就是可变的，所以给一个对象添加额外的功能本身并不是什么问题。

装饰器模式的一个很方便的特性是可以对我们需要的特性进行定制和配置。刚开始时，我们有一个拥有基本功能的对象，然后可以从可用的装饰器中去挑选一些需要用到的去增加这个对象，甚至如果顺序很重要的话，还可以指定增强的顺序。

用法

我们来看一下这个模式的示例用法。假设你正在做一个卖东西的web应用，每个新交易是一个新的sale对象。这个对象“知道”交易的价格并且可以通过调用sale.getPrice()方法返回。根据环境的不同，你可以开始用一些额外的功能来装饰这个对象。假设一个场景是这笔交易是发生在加拿大的一个省Québec，在这种情况下，购买者需要付联邦税和Québec省税。根据装饰器模式的用法，你需要指明使用联邦税装饰器和Québec省税装饰器来装饰这个对象。然后你还可以给这个对象装饰一些价格格式的功能。这个场景的使用方式可能是像这样：

```
var sale = new Sale(100); // the price is 100 dollars
sale = sale.decorate('fedtax'); // add federal tax
sale = sale.decorate('quebec'); // add provincial tax
sale = sale.decorate('money'); // format like money
sale.getPrice(); // "$112.88"
```

在另一种场景下，购买者在一个不需要交省税的省，并且你想用加拿大元的格式来显示价格，你可以这样做：

```
var sale = new Sale(100); // the price is 100 dollars
sale = sale.decorate('fedtax'); // add federal tax
sale = sale.decorate('cdn'); // format using CDN
sale.getPrice(); // "CDN$ 105.00"
```

如你所见，这是一种在运行时很灵活的方法来添加功能和调整对象。我们来看一下如何来实现这种模式。

实现

一种实现装饰器模式的方法是让每个装饰器成为一个拥有应该被重写的方法的对象。每个装饰器实际上是继承自已经被前一个装饰器增强过的对象。装饰器的每个方法都会调用父对象（继承自的对象）的同名方法并取得值，然后做一些额外的处理。

最终的效果就是当你在第一个例子中调用sale.getPrice()时，实际上是在调用money装饰器的方法（图7-1）。但是因为每个装饰器会先调用父对象的方法，money的getPrice()先调用quebec的getPrice()，而它又会去调用fedtax的getPrice()方法，依次类推。这个链会一直走到原始的未经装饰的由Sale()构造函数实现的getPrice()。

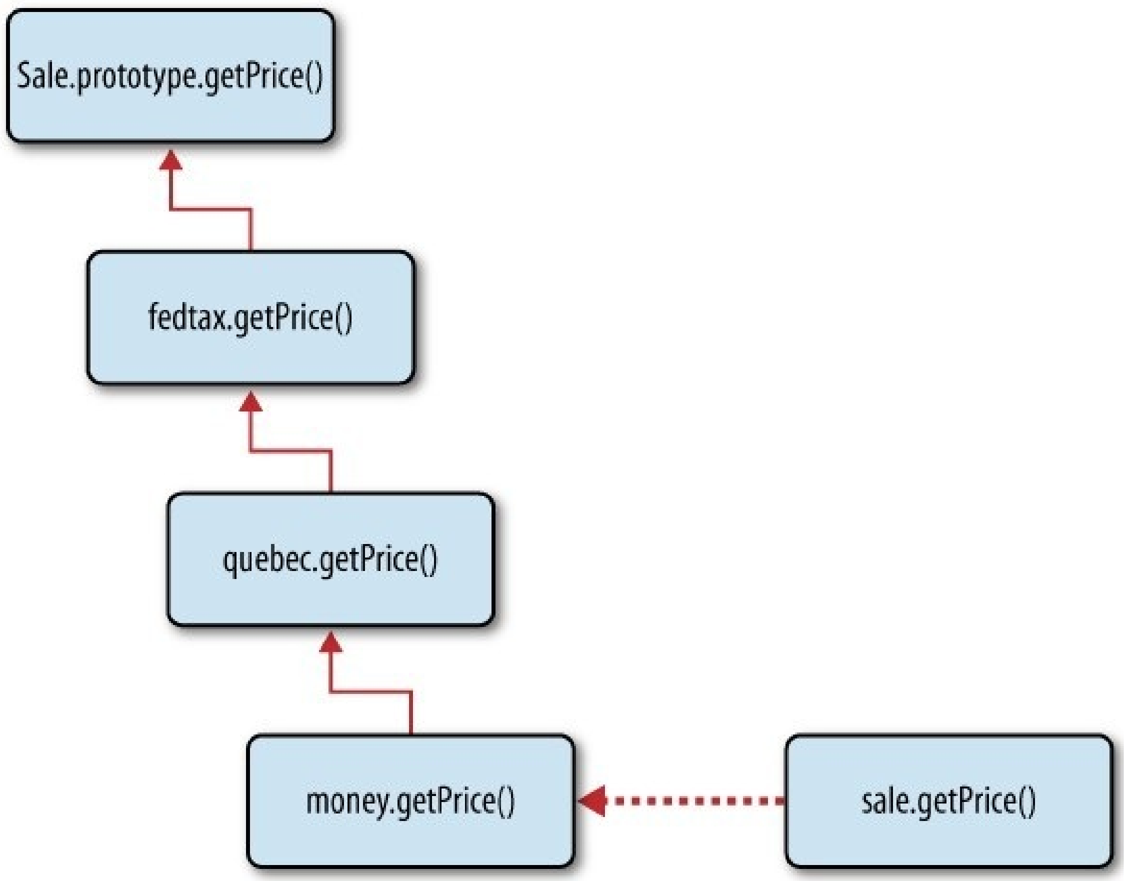


图7-1

装饰器模式的实现

这个实现以一个构造函数和一个原型方法开始：

```
function Sale(price) {
    this.price = price || 100;
}
Sale.prototype.getPrice = function () {
    return this.price;
};
```

装饰器对象将都被作为构造函数的属性实现：

```
Sale.decorators = {};
```

我们来看一个装饰器的例子。这是一个对象，实现了一个自定义的getPrice()方法。注意这个方法首先从父对象的方法中取值然后修改这个值：

```
Sale.decorators.fedtax = {
    getPrice: function () {
        var price = this.uber.getPrice();
        price += price * 5 / 100;
        return price;
    }
};
```

使用类似的方法我们可以实现任意多个需要的其它装饰器。他们的实现方式像插件一样来扩展核心的Sale()的功能。他们甚至可以被放到额外的文件中，被第三方的开发者来开发和共享：

```
Sale.decorators.quebec = {
    getPrice: function () {
        var price = this.uber.getPrice();
        price += price * 7.5 / 100;
        return price;
    }
};

Sale.decorators.money = {
    getPrice: function () {
        return "$" + this.uber.getPrice().toFixed(2);
    }
};

Sale.decorators.cdn = {
    getPrice: function () {
        return "CDN$ " + this.uber.getPrice().toFixed(2);
    }
};
```

最后我们来看decorate()这个神奇的方法，它把所有上面说的片段都串起来了。记得它是这样被调用的：

```
sale = sale.decorate('fedtax');
```

字符串'fedtax'对应在Sale.decorators.fedtax中实现的对象。被装饰过的最新的对象newobj将从现在有的对象（也就是this对象，它要么是原始的对象，要么是经过最后一个装饰器装饰过的对象）中继承。实现这一部分需要用到前面章节中提到的临时构造函数模式。我们也设置一个uber属性给newobj以便子对象可以访问到父对象。然后我们从装饰器中复制所有额外的属性到被装饰的对象newobj中。最后，在我们的例子中，newobj被返回并且成为被更新过的sale对象。

```
Sale.prototype.decorate = function (decorator) {
    var F = function () {},
        overrides = this.constructor.decorators[decorator],
        i, newobj;
    F.prototype = this;
    newobj = new F();
    newobj.uber = F.prototype;
    for (i in overrides) {
        if (overrides.hasOwnProperty(i)) {
            newobj[i] = overrides[i];
        }
    }
    return newobj;
};
```

使用列表实现

我们来看另一个明显不同的实现方法，受益于JavaScript的动态特性，它完全不需要使用继承。同时，我们也可以简单地将前一个方面的结果作为参数传给下一个方法，而不需要每一个方法都去调用前一个方法。

这样的实现方法还允许很容易地反装饰（undecorating）或者撤销一个装饰，这仅仅需要从一个装饰器列表中移除一个条目。

用法示例也会明显简单一些，因为我们不需要将decorate()的返回值赋值给对象。在这个实现中，decorate()不对对象做任何事情，它只是简单地将装饰器加入到一个列表中：

```
var sale = new Sale(100); // the price is 100 dollars
sale.decorate('fedtax'); // add federal tax
sale.decorate('quebec'); // add provincial tax
sale.decorate('money'); // format like money
sale.getPrice(); // "$112.88"
```

Sale()构造函数现在有了一个作为自己属性的装饰器列表：

```
function Sale(price) {
    this.price = (price > 0) || 100;
    this.decorators_list = [];
}
```

可用的装饰器仍然被实现为Sale.decorators的属性。注意getPrice()方法现在更简单了，因为它们不需要调用父对象的getPrice()来获取结果，结果已经作为参数传递给它们了：

```
Sale.decorators = {};

Sale.decorators.fedtax = {
    getPrice: function (price) {
        return price + price * 5 / 100;
    }
};

Sale.decorators.quebec = {
    getPrice: function (price) {
        return price + price * 7.5 / 100;
    }
};

Sale.decorators.money = {
    getPrice: function (price) {
        return "$" + price.toFixed(2);
    }
};
```

最有趣的部分发生在父对象的decorate()和getPrice()方法上。在前一种实现方式中，decorate()还是多少有些复杂，而getPrice()十分简单。在这种实现方式中事情反过来了：decorate()只需要往列表中添加条目而getPrice()做了所有的工作。这些工作包括遍历现在添加的装饰器的列表，然后调用它们的getPrice()方法，并将结果传递给前一个：

```
Sale.prototype.decorate = function (decorator) {
    this.decorators_list.push(decorator);
};

Sale.prototype.getPrice = function () {
    var price = this.price,
        i,
        max = this.decorators_list.length,
        name;
    for (i = 0; i < max; i += 1) {
        name = this.decorators_list[i];
        price = Sale.decorators[name].getPrice(price);
    }
    return price;
};
```

装饰器模式的第二种实现方式更简单一些，并且没有引入继承。装饰的方法也会简单。所有的工作都由“同意”被装饰的方法来做。在这个示例实现中，getPrice()是唯一被允许装饰的方法。如果你想有更多可以被装饰的方法，那遍历装饰器列表的工作就需要由每个方法重复去做。但是，这可以很容易地被抽象到一个辅助方法中，给它传一个方法然后使这个方法“可被装饰”。如果这样实现的话，decorators_list属性就应该是一个对象，它的属性名字是方法名，值是装饰器对象的数组。

策略模式

策略模式允许在运行的时候选择算法。你的代码的使用者可以在处理特定任务的时候根据即将要做的事情的上下文来从一些可用的算法中选择一个。

使用策略模式的一个例子是解决表单验证的问题。你可以创建一个validator对象，有一个validate()方法。这个方法被调用时不用区分具体的表单类型，它总是会返回同样的结果——一个没有通过验证的列表和错误信息。

但是根据具体的需要验证的表单和数据，你代码的使用者可以选择进行不同类别的检查。你的validator选择最佳的策略来处理这个任务，然后将具体的数据检查工作交给合适的算法去做。

数据验证示例

假设你有一个下面这样的数据，它可能来自页面上的一个表单，你希望验证它是不是有效的数据：

```
var data = {
    first_name: "Super",
    last_name: "Man",
    age: "unknown",
    username: "o_O"
};
```

对这个例子中的validator，它需要知道哪个是最佳策略，因此你需要先配置它，给它设定好规则以确定哪些是有效的数据。

假设你不需要姓，名字可以接受任何内容，但要求年龄是一个数字，并且用户名只允许包含字母和数字。配置可能是这样的：

```
validator.config = {
    first_name: 'isNonEmpty',
    age: 'isNumber',
    username: 'isAlphaNum'
};
```

现在validator对象已经有了用来处理数据的配置，你可以调用validate()方法，然后将任何验证错误打印到控制台上：

```
validator.validate(data);
if (validator.hasErrors()) {
    console.log(validator.messages.join("\n"));
}
```

它可能会打印出这样的信息：

```
Invalid value for *age*, the value can only be a valid number, e.g. 1, 3.14 or 2010
Invalid value for *username*, the value can only contain characters and numbers, no speci
```

现在我们来看一下这个validator是如何实现的。所有可用的用来检查的逻辑都是拥有一个validate()方法的对象，它们还有一行辅助信息用来显示错误信息：

```
// checks for non-empty values
validator.types.isNonEmpty = {
    validate: function (value) {
        return value !== "";
    },
    instructions: "the value cannot be empty"
};

// checks if a value is a number
validator.types.isNumber = {
    validate: function (value) {
        return !isNaN(value);
    },
    instructions: "the value can only be a valid number, e.g. 1, 3.14 or 2010"
};

// checks if the value contains only letters and numbers
validator.types.isAlphaNum = {
    validate: function (value) {
        return !/[^a-z0-9]/i.test(value);
    },
    instructions: "the value can only contain characters and numbers, no special symbols"
};
```

最后，validator对象的核心是这样的：

```
var validator = {

    // all available checks
    types: {},

    // error messages in the current
    // validation session
    messages: [],

    // current validation config
    // name: validation type
    config: {},

    // the interface method
    // `data` is key => value pairs
    validate: function (data) {

        var i, msg, type, checker, result_ok;

        // reset all messages
        this.messages = [];
        for (i in data) {

            if (data.hasOwnProperty(i)) {

                type = this.config[i];
                checker = this.types[type];

                if (!type) {
                    continue; // no need to validate
                }
                if (!checker) { // uh-oh
                    throw {
                        name: "ValidationError",
                        message: "No handler to validate type " + type
                    };
                }

                result_ok = checker.validate(data[i]);
                if (!result_ok) {
                    msg = "Invalid value for *" + i + "*, " + checker.instructions;
                    this.messages.push(msg);
                }
            }
        }
        return this.hasErrors();
    },

    // helper
    hasErrors: function () {
        return this.messages.length !== 0;
    }
};
```

如你所见，validator对象是通用的，在所有的需要验证的场景下都可以保持这个样子。改进它的办法就是增加更多类型的检查。如果你将它用在很多页面上，每快你就会有一个非常好的验证类型的集合。然后在每个新的使用场景下你需要做的仅仅是配置validator然后调用validate()方法。

外观模式

外观模式是一种很简单的模式，它只是为对象提供了更多的可供选择的接口。使方法保持短小而不是处理太多的工作是一种很好的实践。在这种实践的指导下，你会有一大堆的方法，而不是一个有着非常多参数的uber方法。有些时候，两个或者更多的方法会经常被一起调用。在这种情况下，创建另一个将这些重复调用包裹起来的方法就变得意义了。

例如，在处理浏览器事件的时候，有以下的事件：

- stopPropagation()

  阻止事件冒泡到父节点

- preventDefault()

  阻止浏览器执行默认动作（如打开链接或者提交表单）

这是两个有不同目的的相互独立的方法，他们也应该被保持独立，但与此同时，他们也经常被一起调用。所以为了不在应用中到处重复调用这两个方法，你可以创建一个外观方法来调用它们：

```
var myevent =  {
    // ...
    stop: function (e) {
        e.preventDefault();
        e.stopPropagation();
    }
    // ...
};
```

外观模式也适用于一些浏览器脚本的场景，即将浏览器的差异隐藏在一个外观方法下面。继续前面的例子，你可以添加一些处理IE中事件API的代码：

```
var myevent =  {
    // ...
    stop: function (e) {
        // others
        if (typeof e.preventDefault === "function") {
            e.preventDefault();
        }
        if (typeof e.stopPropagation === "function") {
            e.stopPropagation();
        }
        // IE
        if (typeof e.returnValue === "boolean") {
            e.returnValue = false;
        }
        if (typeof e.cancelBubble === "boolean") {
            e.cancelBubble = true;
        }
    }
    // ...
};
```

外观模式在做一些重新设计和重构工作时也很有用。当你想用一个不同的实现来替换某个对象的时候，你可能需要工作相当长一段时间（一个复杂的对象），与此同时，一些使用这个新对象的代码也在被同步编写。你可以先想好新对象的API，然后使用新的API创建一个外观

方法在旧的对象前面。使用这种方式，当你完全替换到旧的对象的时候，你只需要修改少量客户代码，因为新的客户代码已经是在使用新的API了。

代理模式

在代理设计模式中，一个对象充当了另一个对象的接口的角色。它和外观模式不一样，外观模式带来的方便仅限于将几个方法调用联合起来。而代理对象位于某个对象和它的客户之间，可以保护对对象的访问。

这个模式看起来开销有点大，但在出于性能考虑时非常有用。代理对象可以作为对象（也叫“真正的主体”）的保护者，让真正的主体对象做尽量少的工作。

一种示例用法是我们称之为“懒初始化”（延迟初始化）的东西。假设初始化真正的主体是开销很大的，并且正好客户代码将它初始化后并不真正使用它。在这种情况下，代理对象可以作为真正的主体的接口起到帮助作用。代理对象接收到初始化请求，但在真正的主体真正被使用之前都不会将它传递过去。

图7-2展示了这个场景，当客户代码发出初始化请求时，代理对象回复一切就绪，但并没有将请求传递过去，只有在客户代码真正需要真正的主体做些工作的时候才将两个请求一起传递过去。

图7-2 通过代理对象时客户代码与真正的主体的关系

一个例子

在真正的主体做某件工作开销很大时，代理模式很有用处。在web应用中，开销最大的操作之一就是网络请求，此时尽可能地合并HTTP请求是有意义的。我们来看一个这种场景下应用代理模式的实例。

一个视频列表（**expando**）

我们假设有一个用来播放选中视频的应用。你可以在这里看到真实的例子http://www.jspatterns.com/book/7/proxy.html。

页面上有一个视频标题的列表，当用户点击视频标题的时候，标题下方的区域会展开并显示视频的更多信息，同时也使得视频可被播放。视频的详细信息和用来播放的URL并不是页面的一部分，它们需要通过网络请求来获取。服务端可以接受多个视频ID，这样我们就可以在合适的时候通过一次请求多个视频信息来减少HTTP请求以加快应用的速度。

我们的应用允许一次展开好几个（或全部）视频，所以这是一个合并网络请求的绝好机会。

图7-3 真实的视频列表

没有代理对象的情况

这个应用中最主要的角色是两个对象：

- videos

  负责对信息区域展开/收起（videos.getInfo()方法）和播放视频的响应（videos.getPlayer()方法）

- http

  负责通过http.makeRequest()方法与服务端通讯

当没有代理对象的时候，videos.getInfo()会为每个视频调用一次http.makeRequest()方法。当我们添加代理对象proxy后，它将位于vidoes和http中间，接手对makeRequest()的调用，并在可能的时候合并请求。

我们首先看一下没有代理对象的代码，然后添加代理对象来提升应用的响应速度。

HTML

HTML代码仅仅是一个链接列表：

```
<p><span id="toggle-all">Toggle Checked</span></p>
<ol id="vids">
    <li><input type="checkbox" checked><a
    href="http://new.music.yahoo.com/videos/--2158073">Gravedigger</a></li>
    <li><input type="checkbox" checked><a
    href="http://new.music.yahoo.com/videos/--4472739">Save Me</a></li>
    <li><input type="checkbox" checked><a
    href="http://new.music.yahoo.com/videos/--45286339">Crush</a></li>
    <li><input type="checkbox" checked><a
    href="http://new.music.yahoo.com/videos/--2144530">Don't Drink The Water</a></li>
    <li><input type="checkbox" checked><a
    href="http://new.music.yahoo.com/videos/--217241800">Funny the Way It Is</a></li>
    <li><input type="checkbox" checked><a
    href="http://new.music.yahoo.com/videos/--2144532">What Would You Say</a></li>
</ol>
```

事件处理

现在我们来看一下事件处理的逻辑。首先我们定义一个方便的快捷函数$：

```
var $ = function (id) {
    return document.getElementById(id);
};
```

使用事件代理（第8章有更多关于这个模式的内容），我们将所有id="vids"的条目上的点击事件统一放到一个函数中处理：

```
$('vids').onclick = function (e) {
    var src, id;

    e = e || window.event;
    src = e.target || e.srcElement;

    if (src.nodeName !== "A") {
        return;
    }

    if (typeof e.preventDefault === "function") {
        e.preventDefault();
    }
    e.returnValue = false;

    id = src.href.split('--')[1];

    if (src.className === "play") {
        src.parentNode.innerHTML = videos.getPlayer(id);
        return;
    }

    src.parentNode.id = "v" + id;
    videos.getInfo(id);
};
```

videos对象

videos对象有三个方法：

- getPlayer()

  返回播放视频需要的HTML代码（跟我们讨论的无关）

- updateList()

  网络请求的回调函数，接受从服务器返回的数据，然后生成用于视频详细信息的HTML代码。这一部分也没有什么太有趣的事情。

- getInfo()

  这个方法切换视频信息的可视状态，同时也调用http对象的方法，并传递updaetList()作为回调函数。

下面是这个对象的代码片段：

```
var videos = {

    getPlayer: function (id) {...},
    updateList: function (data) {...},

    getInfo: function (id) {

        var info = $('info' + id);

        if (!info) {
            http.makeRequest([id], "videos.updateList");
            return;
        }

        if (info.style.display === "none") {
            info.style.display = '';
        } else {
            info.style.display = 'none';
        }

    }
};
```

http对象

http对象只有一个方法，它向Yahoo!的YQL服务发起一个JSONP请求：

```
var http = {
    makeRequest: function (ids, callback) {
        var url = 'http://query.yahooapis.com/v1/public/yql?q=',
            sql = 'select * from music.video.id where ids IN ("%ID%")',
            format = "format=json",
            handler = "callback=" + callback,
            script = document.createElement('script');

        sql = sql.replace('%ID%', ids.join('","'));
        sql = encodeURIComponent(sql);

        url += sql + '&' + format + '&' + handler;
        script.src = url;

        document.body.appendChild(script);
    }
};
```

> YQL（Yahoo! Query Language）是一种web service，它提供了使用类似SQL的语法来调用很多其它web service的能力，使得使用者不需要学习每个service的API。

当所有的六个视频都被选中后，将会向服务端发起六个独立的像这样的YQL请求：

select * from music.video.id where ids IN ("2158073")

代理对象

前面的代码工作得很正常，但我们可以让它工作得更好。proxy对象就在这样的场景中出现，并接管了http和videos对象之间的通讯。它将使用一个简单的逻辑来尝试合并请求：50ms的延迟。videos对象并不直接调用后台接口，而是调用proxy对象的方法。proxy对象在转发这个请求前将会等待一段时间，如果在等待的50ms内有另一个来自videos的调用，则它们将被合

并为同一个请求。50ms的延迟对用户来说几乎是无感知的，但是却可以用来合并请求以提升点击“toggle”时的体验，一次展开多个视频。它也可以显著降低服务器的负载，因为web服务器只需要处理更少量的请求。

合并后查询两个视频信息的YQL大概是这样：

```
select * from music.video.id where ids IN ("2158073", "123456")
```

在修改后的代码中，唯一的变化是videos.getInfo()现在调用的是proxy.makeRequest()而不是http.makeRequest()，像这样：

```
proxy.makeRequest(id, videos.updateList, videos);
```

proxy对象创建了一个队列来收集50ms之内接受到的视频ID，然后将这个队列传递给http对象，并提供回调函数，因为videos.updateList()只能处理一次接收到的数据。（译注：指每次传入的回调函数只能处理当次接收到的数据。）

下面是proxy对象的代码：

```
var proxy = {
    ids: [],
    delay: 50,
    timeout: null,
    callback: null,
    context: null,
    makeRequest: function (id, callback, context) {
        // add to the queue
        this.ids.push(id);

        this.callback = callback;
        this.context  = context;

        // set up timeout
        if (!this.timeout) {
            this.timeout = setTimeout(function () {
                proxy.flush();
            }, this.delay);
        }
    },
    flush: function () {

        http.makeRequest(this.ids, "proxy.handler");

        // clear timeout and queue
        this.timeout = null;
        this.ids = [];

    },
    handler: function (data) {
        var i, max;

        // single video
        if (parseInt(data.query.count, 10) === 1) {
            proxy.callback.call(proxy.context, data.query.results.Video);
            return;
        }

        // multiple videos
        for (i = 0, max = data.query.results.Video.length; i < max; i += 1) {
            proxy.callback.call(proxy.context, data.query.results.Video[i]);
        }
    }
};
```

了解代理模式后就在只简单地改动一下原来的代码的情况下，将多个web service请求合并为一个。

图7-4和7-5展示了使用代理模式将与服务器三次数据交互（不用代理模式时）变为一次交互的过程。

图7-4 与服务器三次数据交互

图7-5 通过一个代理对象合并请求，减少与服务器数据交互

使用代理对象做缓存

在这个例子中，客户对象（videos）已经可以做到不对同一个对象重复发出请求。但现实情况中并不总是这样。这个代理对象还可以通过缓存之前的请求结果到cache属性中来进一步保护真正的主体http对象（图7-6）。然后当videos对象需要对同一个ID的视频请求第二次时，proxy对象可以直接从缓存中取出，从而避免一次网络交互。

图7-6 代理缓存

中介者模式

一个应用不论大小，都是由一些彼此独立的对象组成的。所有的对象都需要一个通讯的方式来保持可维护性，即你可以安全地修改应用的一部分而不破坏其它部分。随着应用的开发和维护，会有越来越多的对象。然后，在重构代码的时候，对象可能会被移除或者被重新安排。当对象知道其它对象的太多信息并且直接通讯（直接调用彼此的方法或者修改属性）时，会导致我们不愿意看到的紧耦合。当对象耦合很紧时，要修改一个对象而不影响其它的对象是很困难的。此时甚至连一个最简单的修改都变得不那么容易，甚至连一个修改需要用多长时间都难以评估。

中介者模式就是一个缓解此问题的办法，它通过解耦来提升代码的可维护性（见图7-7）。在这个模式中，各个彼此合作的对象并不直接通讯，而是通过一个mediator（中介者）对象通讯。当一个对象改变了状态后，它就通知中介者，然后中介者再将这个改变告知给其它应该知道这个变化的对象。

图7-7 中介者模式中的对象关系

中介者示例

我们来看一个使用中介者模式的实例。这个应用是一个游戏，它的玩法是比较两位游戏者在半分钟内按下按键的次数，次数多的获胜。玩家1需要按的是1，玩家2需要按的是0（这样他们的手指不会搅在一起）。当前分数会显示在一个计分板上。

对象列表如下：

- Player 1
- Player 2
- Scoreboard
- Mediator

中介者Mediator知道所有的对象。它与输入设备（键盘）打交道，处理keypress事件，决定现在是哪位玩家玩的，然后通知这个玩家（见图7-8）。玩家负责玩（即给自己的分数加一分），然后通知中介者他这一轮已经玩完。中介者再告知计分板最新的分数，计分板更新显

示。

除了中介者之外，其它的对象都不知道有别的对象存在。这样就使得更新这个游戏变得很简单，比如要添加一位玩家或者是添加另外一个显示剩余时间的地方。

你可以在这里看到这个游戏的在线演示http://jspatterns.com/book/7/mediator.html。

图7-8 游戏涉及的对象

玩家对象是通过Player()构造函数来创建的，有自己的points和name属性。原型上的play()方法负责给自己加一分然后通知中介者：

```
function Player(name) {
    this.points = 0;
    this.name = name;
}
Player.prototype.play = function () {
    this.points += 1;
    mediator.played();
};
```

scoreboard对象（计分板）有一个update()方法，它会在每次玩家玩完后被中介者调用。计分析根本不知道玩家的任何信息，也不保存分数，它只负责显示中介者给过来的分数：

```
var scoreboard = {

    // HTML element to be updated
    element: document.getElementById('results'),

    // update the score display
    update: function (score) {

        var i, msg = '';
        for (i in score) {

            if (score.hasOwnProperty(i)) {
                msg += '<p><strong>' + i + '<\/strong>: ';
                msg += score[i];
                msg += '<\/p>';
            }
        }
        this.element.innerHTML = msg;
    }
};
```

现在我们来看一下mediator对象（中介者）。在游戏初始化的时候，在setup()方法中创建游戏者，然后放后players属性以便后续使用。played()方法会被游戏者在每轮玩完后调用，它更新score哈希然表然后将它传给scoreboard用于显示。最后一个方法是keypress()，负责处理键盘事件，决定是哪位玩家玩的，并且通知它：

```
var mediator = {

    // all the players
    players: {},

    // initialization
    setup: function () {
        var players = this.players;
        players.home = new Player('Home');
        players.guest = new Player('Guest');

    },

    // someone plays, update the score
    played: function () {
        var players = this.players,
        score = {
            Home: players.home.points,
            Guest: players.guest.points
        };

        scoreboard.update(score);
    },

    // handle user interactions
    keypress: function (e) {
        e = e || window.event; // IE
        if (e.which === 49) { // key "1"
            mediator.players.home.play();
            return;
        }
        if (e.which === 48) { // key "0"
            mediator.players.guest.play();
            return;
        }
    }
};
```

最后一件事是初始化和结束游戏：

```
// go!
mediator.setup();
window.onkeypress = mediator.keypress;

// game over in 30 seconds
setTimeout(function () {
    window.onkeypress = null;
    alert('Game over!');
}, 30000);
```

观察者模式

观察者模式被广泛地应用于JavaScript客户端编程中。所有的浏览器事件（mouseover，keypress等）都是使用观察者模式的例子。这种模式的另一个名字叫“自定义事件”，意思是这些事件是被编写出来的，和浏览器触发的事件相对。它还有另外一个名字叫“订阅者/发布者”模式。

使用这个模式的最主要目的就是促进代码触解耦。在观察者模式中，一个对象订阅另一个对象的指定活动并得到通知，而不是调用另一个对象的方法。订阅者也被叫作观察者，被观察的对象叫作发布者或者被观察者（译注：subject，不知道如何翻译，第一次的时候译为“主体”，第二次译时觉得不妥，还是直接叫被观察者好了）。当一个特定的事件发生的时候，发布者会通知（调用）所有的订阅者，同时还可能以事件对象的形式传递一些消息。

例1：杂志订阅

为了理解观察者模式的实现方式，我们来看一个具体的例子。我们假设有一个发布者paper，它发行一份日报和一份月刊。无论是日报还是月刊发行，有一个名叫joe的订阅者都会收到通知。

paper对象有一个subscribers属性，它是一个数组，用来保存所有的订阅者。订阅的过程就仅仅是将订阅者放到这个数组中而已。当一个事件发生时，paper遍历这个订阅者列表，然后通知它们。通知的意思也就是调用订阅者对象的一个方法。因此，在订阅过程中，订阅者需要提供一个方法给paper对象的subscribe()。

paper对象也可以提供unsubscribe()方法，它可以将订阅者从数组中移除。paper对象的最后一个重要的方法是publish()，它负责调用订阅者的方法。总结一下，一个发布者对象需要有这些成员：

- subscribers

  一个数组

- subscribe()

  将订阅者加入数组

- unsubscribe()

  从数组中移除订阅者

- publish()

  遍历订阅者并调用它们订阅时提供的方法

所有三个方法都需要一个type参数，因为一个发布者可能触发好几种事件（比如同时发布杂志和报纸），而订阅者可以选择性地订阅其中的一种或几种。

因为这些成员对任何对象来说都是通用的，因此将它们作为独立对象的一部分提取出来是有意义的。然后，我们可以（通过混元模式）将它们复制到任何一个对象中，将这些对象转换为订阅者。

下面是这些发布者通用功能的一个示例实现，它定义了上面列出来的所有成员，还有一个辅助的visitSubscribers()方法：

```
var publisher = {
    subscribers: {
        any: [] // event type: subscribers
    },
    subscribe: function (fn, type) {
        type = type || 'any';
        if (typeof this.subscribers[type] === "undefined") {
            this.subscribers[type] = [];
        }
        this.subscribers[type].push(fn);
    },
    unsubscribe: function (fn, type) {
        this.visitSubscribers('unsubscribe', fn, type);
    },
    publish: function (publication, type) {
        this.visitSubscribers('publish', publication, type);
    },
    visitSubscribers: function (action, arg, type) {
        var pubtype = type || 'any',
            subscribers = this.subscribers[pubtype],
            i,
            max = subscribers.length;

        for (i = 0; i < max; i += 1) {
            if (action === 'publish') {
                subscribers[i](arg);
            } else {
                if (subscribers[i] === arg) {
                    subscribers.splice(i, 1);
                }
            }
        }
    }
};
```

下面这个函数接受一个对象作为参数，并通过复制通用的发布者的方法将这个对象墨迹成发布者：

```
function makePublisher(o) {
    var i;
    for (i in publisher) {
        if (publisher.hasOwnProperty(i) && typeof publisher[i] === "function") {
            o[i] = publisher[i];
        }
    }
    o.subscribers = {any: []};
}
```

现在我们来实现paper对象，它能做的事情就是发布日报和月刊：

```
var paper = {
    daily: function () {
        this.publish("big news today");
    },
    monthly: function () {
        this.publish("interesting analysis", "monthly");
    }
};
```

将paper对象变成发布者：

```
makePublisher(paper);
```

现在我们有了一个发布者，让我们再来看一下订阅者对象joe，它有两个方法：

```
var joe = {
    drinkCoffee: function (paper) {
        console.log('Just read ' + paper);
    },
    sundayPreNap: function (monthly) {
        console.log('About to fall asleep reading this ' + monthly);
    }
};
```

现在让joe来订阅paper：

```
paper.subscribe(joe.drinkCoffee);
paper.subscribe(joe.sundayPreNap, 'monthly');
```

如你所见，joe提供了一个当默认的any事件发生时被调用的方法，还提供了另一个当monthly事件发生时被调用的方法。现在让我们来触发一些事件：

```
paper.daily();
paper.daily();
paper.daily();
paper.monthly();
```

这些发布行为都会调用joe的对应方法，控制台中输出的结果是：

```
Just read big news today
Just read big news today
Just read big news today
About to fall asleep reading this interesting analysis
```

这里值得称道的地方就是paper对象并没有硬编码写上joe，而joe也同样没有硬编码写上paper。这里也没有知道所有事情的中介者对象。所有涉及到的对象都是松耦合的，而且在不修改代码的前提下，我们可以给paper添加更多的订阅者，同时joe也可以在任何时候取消订阅。

让我们更进一步，将joe也变成一个发布者。（毕竟，在博客和微博上，任何人都可以是发布者。）这样，joe变成发布者之后就可以在Twitter上更新状态：

```
makePublisher(joe);
joe.tweet = function (msg) {
    this.publish(msg);
};
```

现在假设paper的公关部门准备通过Twitter收集读者反馈，于是它订阅了joe，提供了一个方法readTweets()：

```
paper.readTweets = function (tweet) {
    alert('Call big meeting! Someone ' + tweet);
};
joe.subscribe(paper.readTweets);
```

这样每当joe发出消息时，paper就会弹出警告窗口：

```
joe.tweet("hated the paper today");
```

结果是一个警告窗口："Call big meeting! Someone hated the paper today"。

你可以在http://jspatterns.com/book/7/observer.html看到完整的源代码，并且在控制台中运行这个实例。

例2：按键游戏

我们来看另一个例子。我们将实现一个和中介者模式的示例一样的按钮游戏，但这次使用观察者模式。为了让它看起来更高档，我们允许接受无限个玩家，而不限于2个。我们仍然保留用来产生玩家的Player()构造函数，也保留scoreboard对象。只有mediator会变成game对象。

在中介者模式中，mediator对象知道所有涉及到的对象，并且调用它们的方法。而观察者模式中的game对象不是这样，它会让对象来订阅它们感兴趣的事件。比如，scoreboard会订阅game对象的scorechange事件。

首先我们重新看一下通用的publisher对象，并且将它的接口做一点小修改以更贴近浏览器的情况：

- 将publish()，subscribe()，unsubscribe()分别改为fire()，on()，remove()
- 事件的type每次都会被用到，所以把它变成三个方法的第一个参数
- 可以给订阅者的方法额外加一个context参数，以便回调方法可以用this指向它自己所属的对象

新的publisher对象是这样：

```
var publisher = {
    subscribers: {
        any: []
    },
    on: function (type, fn, context) {
        type = type || 'any';
        fn = typeof fn === "function" ? fn : context[fn];

        if (typeof this.subscribers[type] === "undefined") {
            this.subscribers[type] = [];
        }
        this.subscribers[type].push({fn: fn, context: context || this});
    },
    remove: function (type, fn, context) {
        this.visitSubscribers('unsubscribe', type, fn, context);
    },
    fire: function (type, publication) {
        this.visitSubscribers('publish', type, publication);
    },
    visitSubscribers: function (action, type, arg, context) {
        var pubtype = type || 'any',
            subscribers = this.subscribers[pubtype],
            i,
            max = subscribers ? subscribers.length : 0;

        for (i = 0; i < max; i += 1) {
            if (action === 'publish') {
                subscribers[i].fn.call(subscribers[i].context, arg);
            } else {
                if (subscribers[i].fn === arg && subscribers[i].context === context) {
                    subscribers.splice(i, 1);
                }
            }
        }
    }
};
```

新的Player()构造函数是这样：

```
function Player(name, key) {
    this.points = 0;
    this.name = name;
    this.key = key;
    this.fire('newplayer', this);
}

Player.prototype.play = function () {
    this.points += 1;
    this.fire('play', this);
};
```

变动的部分是这个构造函数接受key，代表这个玩家在键盘上用来按之后得分的按键。（这些键预先被硬编码过。）每次创建一个新玩家的时候，一个newplayer事件也会被触发。类似的，每次有一个玩家玩的时候，会触发play事件。

scoreboard对象和原来一样，它只是简单地将当前分数显示出来。

game对象会关注所有的玩家，这样它就可以给出分数并且触发scorechange事件。它也会订阅浏览吕中所有的keypress事件，这样它就会知道按钮对应的玩家：

var game = {

```
keys: {},

addPlayer: function (player) {
    var key = player.key.toString().charCodeAt(0);
    this.keys[key] = player;
},

handleKeypress: function (e) {
    e = e || window.event; // IE
    if (game.keys[e.which]) {
        game.keys[e.which].play();
    }
},

handlePlay: function (player) {
    var i,
        players = this.keys,
        score = {};

    for (i in players) {
        if (players.hasOwnProperty(i)) {
            score[players[i].name] = players[i].points;
        }
    }
    this.fire('scorechange', score);
}
```

};

用于将任意对象转变为订阅者的makePublisher()还是和之前一样。game对象会变成发布者（这样它才可以触发scorechange事件），Player.prototype也会变成发布者，以使得每个玩家对象可以触发play和newplayer事件：

```
makePublisher(Player.prototype);
makePublisher(game);
```

game对象订阅play和newplayer事件（以及浏览器的keypress事件），scoreboard订阅scorechange事件：

```
Player.prototype.on("newplayer", "addPlayer", game);
Player.prototype.on("play", "handlePlay", game);
game.on("scorechange", scoreboard.update, scoreboard);
window.onkeypress = game.handleKeypress;
```

如你所见，on()方法允许订阅者通过函数（scoreboard.update）或者是字符串（"addPlayer"）来指定回调函数。当有提供context（如game）时，才能通过字符串来指定回调函数。

初始化的最后一点工作就是动态地创建玩家对象（以及它们对象的按键），用户想要多少个就可以创建多少个：

```
var playername, key;
while (1) {
    playername = prompt("Add player (name)");
    if (!playername) {
        break;
    }
    while (1) {
        key = prompt("Key for " + playername + "?");
        if (key) {
            break;
        }
    }
    new Player(playername, key);
}
```

这就是游戏的全部。你可以在<http://jspatterns .com/book/7/observer-game.html>看到完整的源代码并且试玩一下。

值得注意的是，在中介者模式中，mediator对象必须知道所有的对象，然后在适当的时机去调用对应的方法。而这个例子中，game对象会显得笨一些（译注：指知道的信息少一些），游戏依赖于对象去观察特写的事件然后触发相应的动作：如scoreboard观察scorechange事件。这使得对象之间的耦合更松了（对象间知道彼此的信息越少越好），而代价则是弄清事件和订阅者之间的对应关系会更困难一些。在这个例子中，所有的订阅行为都发生在代码中的同一个地方，而随着应用规模的境长，on()可能会被在各个地方调用（如在每个对象的初始化代码中）。这使得调试更困难一些，因为没有一个集中的地方来看这些代码并理解正在发生什么事情。在观察者模式中，你将不再能看到那种从开头一直跟到结尾的顺序执行方式。

小结

在这章中你学习到了若干种流行的设计模式，并且也知道了如何在JavaScript中实现它们。我们讨论过的设计模式有：

- 单例模式

  只创建类的唯一一个实例。我们看了好几种可以不通过构造函数和类Java语法达成单例的方法。从另一方面来说，JavaScript中所有的对象都是单例。有时候开发者说的单例是指通过模块化模式创建的对象。

- 工厂模式

  一种在运行时通过指定字符串来创建指定类型对象的方法。

- 遍历模式

  通过提供API来实现复杂的自定义数据结构中的遍历和导航。

- 装饰模式

  在运行时通过从预先定义好的装饰器对象来给被装饰对象动态添加功能。

- 策略模式

  保持接口一致的情况下选择最好的策略来完成特写类型的任务。

- 外观模式

  通过包装通用的（或者设计得很差的）方法来提供一个更方便的API。

- 代理模式

  包装一个对象以控制对它的访问，通过合并操作或者是只在眞正需要时执行来尽量避免开销太大的操作。

- 中介者模式

  通过让对象不彼此沟通，只通过一个中介者对象沟通的方法来促进解耦。

- 观察者模式

  通过创建“可被观察的对象”使它在某个事件发生时通知订阅者的方式来解耦。（也叫“订阅者/发布者”或者“自定义事件”。）

DOM和浏览器中的模式

在本书的前面几章中，我们主要关注了JavaScript核心（ECMAScript），并没有涉及太多关于在浏览器中使用JavaScript的内容。在本章，我们将探索一些在浏览器环境中的模式，因为这是最常见的JavaScript程序环境。浏览器脚本编程也是大部分不喜欢JavaScript的人对这门语言的认知。这当然是可以理解，因为在浏览器中有非常多不一致的宿主对象和DOM实现。很明显，任何能够减轻客户端脚本编程的痛楚的最佳初中都是大有益处的。

在本章中，你会看到一些零散的模式，包括DOM编程、事件处理、远程脚本、页面脚本的加载策略以及将JavaScript部署到生产环境的步骤。

但首先，让我们来简要讨论一下如何做客户端脚本编程。

分离

在web应用开发中主要关注的有三种东西：

- 内容

  即HTML文档

- 表现

  指定文档样式的CSS

- 行为

  JavaScript，用来处理用户交互和页面的动态变化

尽可能地将这三者分离可以加强应用在各种用户代理（译注：user agent，即为用户读取页面并呈现的软件，一般指浏览器）的可到达性（译注：delivery，指可被用户代理接受并理解的程度），比如图形浏览器、纯文本浏览器、用于残障人士的辅助技术、移动设备等等。分离常常是和渐进增强的思想一起实现的，我们从一个给最简单的用户代理的最基础的体验（纯HTML）开始，当用户代理的兼容性提升时再添加更多的可以为体验加分的东西。如果浏览器支持CSS，那么用户会看到文档更好的呈现。如果浏览器支持JavaScript，那文档会更像一个应用，提供更多的特性来增强用户体验。

在实践中，分离意味者：

- 在关掉CSS的情况下测试页面，看页面是否仍然可用，内容是否可以呈现和阅读
- 在关掉JavaScript的情况下测试页面，确保页面仍然可以完成它的主要功能，所有的链接都可以正常工作（没有href="#"的链接），表单仍然可以正常填写和提交
- 不要使用内联的事件处理（如onclick）或者是内联的style属性，因为它们不属于内容层

- 使用语义化的HTML元素，比如头部和列表等

JavaScript（行为）层的地位不应该很显赫，也就是说它不应该成为页面正常工作必须的东西，不应该使得用户在使用不支持的浏览器操作时存在障碍。它只应该被用来增强页面。

通常比较优雅的用来处理浏览器差异的方法是特性检测。它的思想是你不应该使用浏览器类型检测来决定代码的逻辑，而是应该检测在当前环境中你需要使用的某个方法或者是属性是否存在。浏览器检测一般认为是一种“反模式”（译注：anitpattern，指不好的模式）。虽然有的情况下不可避免要使用，但它应该是最后考虑的选择，并且应该只在特性检测没有办法给出明确答案（或者造成明显性能问题）的时候使用：

```
// antipattern
if (navigator.userAgent.indexOf('MSIE') !== −1) {
    document.attachEvent('onclick', console.log);
}

// better
if (document.attachEvent) {
    document.attachEvent('onclick', console.log);
}

// or even more specific
if (typeof document.attachEvent !== "undefined") {
    document.attachEvent('onclick', console.log);
}
```

分离也有助于开发、维护，减少升级一个现有应用的难度，因为当出现问题的时候，你知道去看哪一块。当出现一个JavaScript错误的时候，你不需要去看HTML或者是CSS就能修复它。

DOM编程

操作页面的DOM树是在客户端JavaScript编程中最普遍的动作。这也是导致开发者头疼的最主要原因（这也导致了JavaScript名声不好），因为DOM方法在不同的浏览器中实现得有很多差异。这也是为什么使用一个抽象了浏览器差异的JavaScript库能显著提高开发速度的原因。

我们来看一些在访问和修改DOM树时推荐的模式，主要考虑点是性能方面。

DOM访问

DOM操作性能不好，这是影响JavaScript性能的最主要原因。性能不好是因为浏览器的DOM实现通常是和JavaScript引擎分离的。从浏览器的角度来讲，这样做是很有意义的，因为有可能一个JavaScript应用根本不需要DOM，而除了JavaScript之外的其它语言（如IE的VBScript）也可以用来操作页面中的DOM。

一个原则就是DOM访问的次数应该被减少到最低，这意味者：

- 避免在环境中访问DOM
- 将DOM引用赋给本地变量，然后操作本地变量
- 当可能的时候使用selectors API
- 遍历HTML collections时缓存length（见第2章）

看下面例子中的第二个（better）循环，尽管它看起来更长一些，但却要快上几十上百倍（取决于具体浏览器）：

```
// antipattern
for (var i = 0; i < 100; i += 1) {
    document.getElementById("result").innerHTML += i + ", ";
}

// better - update a local variable var i, content = "";
for (i = 0; i < 100; i += 1) {
    content += i + ",";
}
document.getElementById("result").innerHTML += content;
```

在下一个代码片段中，第二个例子（使用了本地变量style）更好，尽管它需要多写一行代码，还需要多定义一个变量：

```
// antipattern
var padding = document.getElementById("result").style.padding,
    margin = document.getElementById("result").style.margin;

// better
var style = document.getElementById("result").style,
    padding = style.padding,
    margin = style.margin;
```

使用selectors API是指使用这个方法：

```
document.querySelector("ul .selected");
document.querySelectorAll("#widget .class");
```

这两个方法接受一个CSS选择器字符串，返回匹配这个选择器的DOM列表（译注：querySelector只返回第一个匹配的DOM）。selectors API在现代浏览器（以及IE8+）可用，它总是会比你使用其它DOM方法来做同样的选择要快。主流的JavaScript库的最近版本都已经使用了这个API，所以你有理由去检查你的项目，确保使用的是最新版本。

给你经常访问的元素加上一个id属性也是有好处的，因为document.getElementById(myid)是找到一个DOM元素最容易也是最快的方法。

DOM操作

除了访问DOM元素之外，你可能经常需要改变它们、删除其中的一些或者是添加新的元素。更新DOM会导致浏览器重绘（repaint）屏幕，也经常导致重排(reflow)（重新计算元素的位置），这些操作代价是很高的。

再说一次，通用的原则仍然是尽量少地更新DOM，这意味着我们可以将变化集中到一起，然后在“活动的”（live）文档树之外去执行这些变化。

当你需要添加一棵相对较大的子树的时候，你应该在完成这棵树的构建之后再放到文档树中。为了达到这个目的，你可以使用文档碎片（document fragment）来包含你的节点。

不要这样添加节点：

```
// antipattern
// appending nodes as they are created

var p, t;

p = document.createElement('p');
t = document.createTextNode('first paragraph');
p.appendChild(t);
document.body.appendChild(p);

p = document.createElement('p');
t = document.createTextNode('second paragraph');
p.appendChild(t);
document.body.appendChild(p);
```

一个更好的版本是创建一个文档碎片，然后“离线地”（译注：即不在文档树中）更新它，当它准备好之后再将它加入文档树中。当你将文档碎片添加到DOM树中时，碎片的内容将会被添加进去，而不是碎片本身。这个特性非常好用。所以当有好几个没有被包裹在同一个父元素的节点时，文档碎片是一个很好的包裹方式。

下面是使用文档碎片的例子：

```
var p, t, frag;

frag = document.createDocumentFragment();

p = document.createElement('p');
t = document.createTextNode('first paragraph');
p.appendChild(t);
frag.appendChild(p);

p = document.createElement('p');
t = document.createTextNode('second paragraph');
p.appendChild(t);
frag.appendChild(p);

document.body.appendChild(frag);
```

这个例子和前面例子中每段更新一次相比，文档树只被更新了一下，只导致一次重排/重绘。

当你添加新的节点到文档中时，文档碎片很有用。当你需要更新已有的节点时，你也可以将这些变化集中。你可以将你要修改的子树的父节点克隆一份，然后对克隆的这份做修改，完成之后再去替换原来的元素。

```
var oldnode = document.getElementById('result'),
    clone = oldnode.cloneNode(true);

// work with the clone...

// when you're done:
oldnode.parentNode.replaceChild(clone, oldnode);
```

事件

在浏览器脚本编程中，另一块充满兼容性问题并且带来很多不愉快的区域就是浏览器事件，比如click，mouseover等等。同样的，一个JavaScript库可以解决支持IE（9以下）和W3C标准实现的双倍工作量。

我们来看一下一些主要的点，因为你在做一些简单的页面或者快速开发的时候可能不会使用已有的库，当然，也有可能你正在写你自己的库。

事件处理

麻烦是从给元素绑定事件开始的。假设你有一个按钮，点击它的时候增加计数器的值。你可以添加一个内联的onclick属性，这在所有的浏览器中都能正常工作，但是会违反分离和渐进增强的思想。所以你应该尽力在JavaScript中来做绑定，而不是在标签中。

假设你有下面的标签：

```
<button id="clickme">Click me: 0</button>
```

你可以将一个函数赋给节点的onclick属性，但你只能这样做一次：

```
// suboptimal solution
var b = document.getElementById('clickme'),
    count = 0;
b.onclick = function () {
    count += 1;
    b.innerHTML = "Click me: " + count;
};
```

如果你希望在按钮点击的时候执行好几个函数，那么在维持松耦合的情况下就不能用这种方法来做绑定。从技术上讲，你可以检测onclick是否已经包含一个函数，如果已经包含，就将它加到你自己的函数中，然后替换onclick的值为你的新函数。但是一个更干净的解决方案是使用addEventListener()方法。这个方法在IE8及以下版本中不存在，在这些浏览器需要使用attachEvent()。

当我们回头看条件初始化模式（第4章）时，会发现一个示例实现是一个很好的解决跨浏览器事件监听的套件。现在我们不讨论细节，只看一下如何给我们的按钮绑定事件：

```
var b = document.getElementById('clickme');
if (document.addEventListener) { // W3C
    b.addEventListener('click', myHandler, false);
} else if (document.attachEvent) { // IE
    b.attachEvent('onclick', myHandler);
} else { // last resort
    b.onclick = myHandler;
}
```

现在当按钮被点击时，myHandler会被执行。让我们来让这个函数实现增加按钮文字"Click me: 0"中的数字的功能。为了更有趣一点，我们假设有好几个按钮，一个myHandler()函数来处理所有的按钮点击。如果我们可以从每次点击的事件对象中获取节点和节点对应的计数器值，那为每个按钮保持一个引用和计数器就显得不高效了。

我们先看一下解决方案，稍后再来做些评论：

```
function myHandler(e) {

    var src, parts;

    // get event and source element
    e = e || window.event;
    src = e.target || e.srcElement;

    // actual work: update label
    parts = src.innerHTML.split(": ");
    parts[1] = parseInt(parts[1], 10) + 1;
    src.innerHTML = parts[0] + ": " + parts[1];

    // no bubble
    if (typeof e.stopPropagation === "function") {
    e.stopPropagation();
    }
    if (typeof e.cancelBubble !== "undefined") {
    e.cancelBubble = true;
    }

    // prevent default action
    if (typeof e.preventDefault === "function") {
    e.preventDefault();
    }
    if (typeof e.returnValue !== "undefined") {
    e.returnValue = false;
    }

}
```

一个在线的例子可以在http://jspatterns.com/book/8/click.html找到。

在这个事件处理函数中，有四个部分：

- 首先，我们需要访问事件对象，它包含事件的一些信息以及触发这个事件的页面元素。事件对象会被传到事件处理回调函数中，但是使用onclick属性时需要使用全局属性window.event来获取。
- 第二部分是真正用于更新文字的部分
- 接下来是阻止事件冒泡。在这个例子中它不是必须的，但通常情况下，如果你不阻止的话，事件会一直冒泡到文档根元素甚至window对象。同样的，我们也需要用两种方法来

阻止冒泡：W3C标准方式（stopPropagation()）和IE的方式（使用cancelBubble）

- 最后，如果需要的话，阻止默认行为。有一些事件（点击链接、提交表单）有默认的行为，但你可以使用preventDefault()（IE是通过设置returnValue的值为false的方式）来阻止这些默认行为。

如你所见，这里涉及到了很多重复性的工作，所以使用第7章讨论过的外观模式创建自己的事件处理套件是很有意义的。

事件委托

事件委托是通过事件冒泡来实现的，它可以减少分散到各个节点上的事件处理函数的数量。如果有10个按钮在一个div元素中，你可以给div绑定一个事件处理函数，而不是给每个按钮都绑定一个。

我们来的睦一个实例，三个按钮放在一个div元素中（图8-1）。你可以在http://jspatterns.com/book/8/click-delegate.html看到这个事件委托的实例。

图8-1 事件委托示例：三个在点击时增加计数器值的按钮

结构是这样的：

```
<div id="click-wrap">
    <button>Click me: 0</button>
    <button>Click me too: 0</button>
    <button>Click me three: 0</button>
</div>
```

你可以给包裹按钮的div绑定一个事件处理函数，而不是给每个按钮绑定一个。然后你可以使用和前面的示例中一样的myHandler()函数，但需要修改一个小地方：你需要将你不感兴趣的点击排除掉。在这个例子中，你只关注按钮上的点击，而在同一个div中产生的其它的点击应该被忽略掉。

myHandler()的改变就是检查事件来源的nodeName是不是“button”:

```
// ...
// get event and source element
e = e || window.event;
src = e.target || e.srcElement;

if (src.nodeName.toLowerCase() !== "button") {
    return;
}
// ...
```

事件委托的坏处是筛选容器中感兴趣的事件使得代码看起来更多了，但好处是性能的提升和更干净的代码，这个好处明显大于坏处，因此这是一种强烈推荐的模式。

主流的JavaScript库通过提供方便的API的方式使得使用事件委托变得很容易。比如YUI3中有Y.delegate()方法，它允许你指定一个用来匹配包裹容器的CSS选择器和一个用于匹配你感兴趣的节点的CSS选择器。这很方便，因为如果事件发生在你不关心的元素上时，你的事件处理回调函数不会被调用。在这种情况下，绑定一个事件处理函数很简单：

```
Y.delegate('click', myHandler, "#click-wrap", "button");
```

感谢YUI抽象了浏览器的差异，已经处理好了事件的来源，使得回调函数更简单了：

```
function myHandler(e) {

    var src = e.currentTarget,
        parts;

    parts = src.get('innerHTML').split(": ");
    parts[1] = parseInt(parts[1], 10) + 1;
    src.set('innerHTML', parts[0] + ": " + parts[1]);

    e.halt();
}
```

你可以在http://jspatterns.com/book/8/click-y-delegate.html看到实例。

长时间运行的脚本

你可能注意到过，有时候浏览器会提示脚本运行时间过长，询问用户是否要停止执行。这种情况你当然不希望发生在自己的应用中，不管它有多复杂。

同时，如果脚本运行时间太长的话，浏览器的UI将变得没有响应，用户不能点击任何东西。这是一种很差的用户体验，应该尽量避免。

在JavaScript中没有线程，但你可以在浏览器中使用setTimeout来模拟，或者在现代浏览器中使用web workers。

setTimeout()

它的思想是将一大堆工作分解成为一小段一小段，然后每隔1毫秒运行一段。使用1毫秒的延迟会导致整个任务完成得更慢，但是用户界面会保持可响应状态，用户会觉得浏览器没有失控，觉得更舒服。

> 1毫秒（甚至0毫秒）的延迟执行命令在实际运行的时候会延迟更多，这取决于浏览器和操作系统。设定0毫秒的延迟并不意味着马上执行，而是指“尽快执行”。比如，在IE中，最短的延迟是15毫秒。

Web Workers

现代浏览器为长时间运行的脚本提供了另一种解决方案：web workers。web workers在浏览器内部提供了后台线程支持，你可以将计算量很大的部分放到一个单独的文件中，比如my_web_worker.js，然后从主程序（页面）中这样调用它：

```
var ww = new Worker('my_web_worker.js');
ww.onmessage = function (event) {
    document.body.innerHTML +=
        "<p>message from the background thread: " + event.data + "</p>";
};
```

下面展示了一个做1亿次简单的数学运算的web worker：

```
var end = 1e8, tmp = 1;

postMessage('hello there');

while (end) {
    end -= 1;
    tmp += end;
    if (end === 5e7) { // 5e7 is the half of 1e8
        postMessage('halfway there, `tmp` is now ' + tmp);
    }
}

postMessage('all done');
```

web worker使用postMessage()来和调用它的程序通讯，调用者通过onmessage事件来接受更新。onmessage事件处理函数接受一个事件对象作为参数，这个对象含有一个由web worker传过来data属性。类似的，调用者（在这个例子中）也可以使用ww.postMessage()来给web worker传递数据，web worker可以通过一个onmessage事件处理函数来接受这些数据。

上面的例子会在浏览器中打印出：

```
message from the background thread: hello there
message from the background thread: halfway there, `tmp` is now 3749999975000001 message
```

远程脚本编程

现代web应用经常会使用远程脚本编程和服务器通讯，而不刷新当前页面。这使得web应用更灵活，更像桌面程序。我们来看一下几种用JavaScript和服务器通讯的方法。

XMLHttpRequest

现在，XMLHttpRequest是一个特别的对象（构造函数），绝大多数浏览器都可以用，它使得我们可以从JavaScript来发送HTTP请求。发送一个请求有以下三步：

1. 初始化一个XMLHttpRequest对象（简称XHR）
2. 提供一个回调函数，供请求对象状态改变时调用

3. 发送请求

第一步很简单：

```
var xhr = new XMLHttpRequest();
```

但是在IE7之前的版本中，XHR的功能是使用ActiveX对象实现的，所以需要做一下兼容处理。

第二步是给readystatechange事件提供一个回调函数：

```
xhr.onreadystatechange = handleResponse;
```

最后一步是使用open()和send()两个方法触发请求。open()方法用于初始化HTTP请求的方法（如GET，POST）和URL。send()方法用于传递POST的数据，如果是GET方法，则是一个空字符串。open()方法的最后一个参数用于指定这个请求是不是异步的。异步是指浏览器在等待响应的时候不会阻塞，这明显是更好的用户体验，因此除非必须要同步，否则异步参数应该使用true：

```
xhr.open("GET", "page.html", true);
xhr.send();
```

下面是一个完整的示例，它获取新页面的内容，然后将当前页面的内容替换掉（可以在<http://jspatterns.com/ book/8/xhr.html>看到示例）：

```
var i, xhr, activeXids = [
    'MSXML2.XMLHTTP.3.0',
    'MSXML2.XMLHTTP',
    'Microsoft.XMLHTTP'
];

if (typeof XMLHttpRequest === "function") { // native XHR
    xhr = new XMLHttpRequest();
} else { // IE before 7
    for (i = 0; i < activeXids.length; i += 1) {
        try {
            xhr = new ActiveXObject(activeXids[i]);
            break;
        } catch (e) {}
    }
}

xhr.onreadystatechange = function () {
    if (xhr.readyState !== 4) {
        return false;
    }
    if (xhr.status !== 200) {
        alert("Error, status code: " + xhr.status);
        return false;
    }
    document.body.innerHTML += "<pre>" + xhr.responseText + "<\/pre>"; };

xhr.open("GET", "page.html", true);
xhr.send("");
```

代码中的一些说明：

- 因为IE6及以下版本中，创建XHR对象有一点复杂，所以我们通过一个数组列出ActiveX的名字，然后遍历这个数组，使用try-catch块来尝试创建对象。
- 回调函数会检查xhr对象的readyState属性。这个属性有0到4一共5个值，4代表“complete”（完成）。如果状态还没有完成，我们就继续等待下一次readystatechange事件。
- 回调函数也会检查xhr对象的status属性。这个属性和HTTP状态码对应，比如200（OK）或者是404（Not found）。我们只对状态码200感兴趣，而将其它所有的都报为错误（为了简化示例，否则需要检查其它不代表出错的状态码）。
- 上面的代码会在每次创建XHR对象时检查一遍支持情况。你可以使用前面提到过的模式（如条件初始化）来重写上面的代码，使得只需要做一次检查。

JSONP

JSONP（JSON with padding）是另一种发起远程请求的方式。与XHR不同，它不受浏览器同源策略的限制，所以考虑到加载第三方站点的安全影响的问题，使用它时应该很谨慎。

一个XHR请求的返回可以是任何类型的文档：

- XML文档（过去很常用）
- HTML片段（很常用）
- JSON数据（轻量、方便）
- 简单的文本文件及其它

使用JSONP的话，数据经常是被包裹在一个函数中的JSON，函数名称在请求的时候提供。

JSONP的请求URL通常是像这样：

```
http://example.org/getdata.php?callback=myHandler
```

getdata.php可以是任何类型的页面或者脚本。callback参数指定用来处理响应的JavaScript函数。

这个URL会被放到一个动态生成的\元素中，像这样：

```
var script = document.createElement("script");
script.src = url;
document.body.appendChild(script);
```

服务器返回一些作为参数传递给回调函数的JSON数据。最终的结果实际上是页面中多了一个新的脚本，这个脚本的内容就是一个函数调用，如：

```
myHandler({"hello": "world"});
```

（译注：原文这里说得不是太明白。JSONP的返回内容如上面的代码片段，它的工作原理是在页面中动态插入一个脚本，这个脚本的内容是函数调用+JSON数据，其中要调用的函数是在页面中已经定义好的，数据以参数的形式存在。一般情况下数据由服务端动态生成，而函数由页面生成，为了使返回的脚本能调用到正确的函数，在请求的时候一般会带上callback参数以便后台动态返回处理函数的名字。）

JSONP示例：井字棋

我们来看一个使用JSONP的井字棋游戏示例，玩家就是客户端（浏览器）和服务器。它们两者都会产生1到9之间的随机数，我们使用JSONP去取服务器产生的数字（图8-2）。

你可以在http://jspatterns.com/book/8/ttt.html玩这个游戏。

图8-2 使用JSONP的井字棋游戏

界面上有两个按钮：一个用于开始新游戏，一个用于取服务器下的棋（客户端下的棋会在一定数量的延时之后自动进行）：

```
<button id="new">New game</button>
<button id="server">Server play</button>
```

界面上包含9个单元格，每个都有对应的id，比如：

```
<td id="cell-1"> </td>
<td id="cell-2"> </td>
<td id="cell-3"> </td>
...
```

整个游戏是在一个全局对象ttt中实现：

```
var ttt = {
    // cells played so far
    played: [],

    // shorthand
    get: function (id) {
        return document.getElementById(id);
    },

    // handle clicks
    setup: function () {
        this.get('new').onclick = this.newGame;
        this.get('server').onclick = this.remoteRequest;
    },

    // clean the board
    newGame: function () {
        var tds = document.getElementsByTagName("td"),
            max = tds.length,
            i;
        for (i = 0; i < max; i += 1) {
            tds[i].innerHTML = " ";
        }
        ttt.played = [];
    },

    // make a request
    remoteRequest: function () {
        var script = document.createElement("script");
        script.src = "server.php?callback=ttt.serverPlay&played=" + ttt.played.join(',');
        document.body.appendChild(script);
    },

    // callback, server's turn to play
    serverPlay: function (data) {
        if (data.error) {
            alert(data.error);
            return;
        }

        data = parseInt(data, 10);
        this.played.push(data);

        this.get('cell-' + data).innerHTML = '<span class="server">X<\/span>';

        setTimeout(function () {
            ttt.clientPlay();
        }, 300); // as if thinking hard
    },

    // client's turn to play
    clientPlay: function () {
        var data = 5;

        if (this.played.length === 9) {
            alert("Game over");
            return;
        }

        // keep coming up with random numbers 1-9
        // until one not taken cell is found
        while (this.get('cell-' + data).innerHTML !== " ") {
            data = Math.ceil(Math.random() * 9);
        }
        this.get('cell-' + data).innerHTML = 'O';
        this.played.push(data);
    }
};
```

ttt对象维护着一个已经填过的单元格的列表ttt.played，并且将它发送给服务器，这样服务器就可以返回一个没有玩过的数字。如果有错误发生，服务器会像这样响应：

```
ttt.serverPlay({"error": "Error description here"});
```

如你所见，JSONP中的回调函数必须是公开的并且全局可访问的函数，它并不一定要是全局函数，也可以是一个全局对象的方法。如果没有错误发生，服务器将会返回一个函数调用，像这样：

```
ttt.serverPlay(3);
```

这里的3是指3号单元格是服务器要下棋的位置。在这种情况下，数据非常简单，甚至都不需要使用JSON格式，只需要一个简单的值就可以了。

框架（**frame**）和图片信标**(image beacon)**

另外一种做远程脚本编程的方式是使用框架。你可以使用JavaScript来创建框架并改变它的src URL。新的URL可以包含数据和函数调用来更新调用者，也就是框架之外的父页面。

远程脚本编程中最最简单的情况是你只需要传递一点数据给服务器，而并不需要服务器的响应内容。在这种情况下，你可以创建一个新的图片，然后将它的src指向服务器的脚本：

```
new Image().src = "http://example.org/some/page.php";
```

这种模式叫作图片信标，当你想发送一些数据给服务器记录时很有用，比如做访问统计。因为信标的响应对你来说完全是没有用的，所以通常的做法（不推荐）是让服务器返回一个1x1的GIF图片。更好的做法是让服务器返回一个“204 No Content”HTTP响应。这意味着返回给客户端的响应只有响应头（header）而没有响应体（body）。

部署**JavaScript**

在生产环境中使用JavaScript时，有不少性能方面的考虑。我们来讨论一下最重要的一些。如果需要了解所有的细节，可以参见O'Reilly出社的《高性能网站建设指南》和《高性能网站建设进阶指南》。

合并脚本

创建高性能网站的第一个原则就是尽量减少外部引用的组件（译注：这里指文件），因为HTTP请求的代价是比较大的。具体就JavaScript而言，可以通过合并外部脚本来显著提高页面加载速度。

我们假设你的页面正在使用jQuery库，这是一个.js文件。然后你使用了一些jQuery插件，这些插件也是单独的文件。这样的话在你还一行代码都没有写的时候就已经有了四五个文件了。把这些文件合并起来是很有意义的，尤其是其中的一些体积很小（2-3kb）时，这种情况下，HTTP协议中的开销会比下载本身还大。合并脚本的意思就是简单地创建一个新的js文件，然后把每个文件的内容粘贴进去。

当然，合并的操作应该放在代码部署到生产环境之前，而不是在开发环境中，因为这会使调试变得困难。

合并脚本的不便之处是：

- 在部署前多了一步操作，但这很容易使用命令行自动化工具来做，比如使用Linux/Unix的cat：

  ```
  $ cat jquery.js jquery.quickselect.js jquery.limit.js > all.js
  ```

- 失去一些缓存上的便利——当你对某个文件做了一点小修改之后，会使得整个合并后的代码缓存失效。所以比较好的方法是为大的项目设定一个发布计划，或者是将代码合并为两个文件：一个包含可能会经常变更的代码，另一个包含那些不会轻易变更的“核心”。
- 你需要处理合并后文件的命名或者是版本问题，比如使用一个时间戳all_20100426.js或者是使用文件内容的hash值。

这就是主要的不便之处，但它带来的好处却是远远大于这些麻烦的。

压缩代码

第二章中，我们讨论过代码压缩。部署之前进行代码压缩也是一个很重要的步骤。

从用户的角度来想，完全没有必要下载代码中的注释，因为这些注释根本不影响代码运行。

压缩代码带来的好处多少取决于代码中注释和空白的数量，也取决于你使用的压缩工具。平均来说，压缩可以减少50%左右的体积。

服务端脚本压缩也是应该要做的事情。配置启用gzip压缩是一个一次性的工作，能带来立杆见影的速度提升。即使你正在使用共享的空间，供应商并没有提供那么多服务器配置的空间，大部分的供应商也会允许使用.htaccess配置文件。所以可以将这些加入到站点根目录的.htaccess文件中：

```
AddOutputFilterByType DEFLATE text/html text/css text/plain text/xml application/javascri
```

平均下来压缩会节省70%的文件体积。将代码压缩和服务端压缩合计起来，你可以期望你的用户只下载你写出来的未压缩文件体积的15%。

缓存头

与流行的观点相反，文件在浏览器缓存中的时间并没有那么久。你可以尽你自己的努力，通过使用Expires头来增加非首次访问时命中缓存的概率：

这也是一个在.htaccess中做的一次性配置工作：

```
ExpiresActive On
ExpiresByType application/x-javascript "access plus 10 years"
```

它的弊端是当你想更改这个文件时，你需要给它重命名，如果你已经处理好了合并的文件命名规则，那你就已经处理好这里的命名问题了。

使用CDN

CDN是指“文件分发网络”（Content Delivery Network）。这是一项收费（有时候还相当昂贵）的托管服务，它将你的文件分发到世界上各个不同的数据中心，但代码中的URL却都是一样的，这样可以使用户更快地访问。

即使你没有CDN的预算，你仍然有一些可以免费使用的东西：

- Google托管了很多流行的开源库，你可以免费使用，并从它的CDN中得到速度提升（译注：鉴于Google在国内的尴尬处境，不建议使用）
- 微软托管了jQuery和自家的Ajax库
- 雅虎在自己的CDN上托管了YUI库

加载策略

怎样在页面上引入脚本，这第一眼看起来是一个简单的问题——使用\元素，然后要么写内联的JavaScript代码或者是在src属性中指定一个独立的文件：

```
// option 1
<script>
console.log("hello world"); </script>
// option 2
<script src="external.js"></script>
```

但是，当你的目标是要构建一个高性能的web应用的时候，有些模式和考虑点还是应该知道的。

作为题外话，来看一些比较常见的开发者会用在\元素上的属性：

- language="JavaScript"

还有一些不同大小写形式的“JavaScript”，有的时候还会带上一个版本号。language属性不应该被使用，因为默认的语言就是JavaScript。版本号也不像想象中工作得那么好，这应该是一个设计上的错误。

- type="text/javascript"

  这个属性是HTML4和XHTML1标准所要求的，但它不应该存在，因为浏览器会假设它就是JavaScript。HTML5不再要求这个属性。除非是要强制通过难，否则没有任何使用type的理由。

- defer

  （或者是HTML5中更好的async）是一种指定浏览器在下载外部脚本时不阻塞页面其它部分的方法，但还没有被广泛支持。关于阻塞的更多内容会在后面提及。

\元素的位置

script元素会阻塞页面的下载。浏览器会同时下载好几个组件（文件），但遇到一个外部脚本的时候，会停止其它的下载，直到脚本文件被下载、解析、执行完毕。这会严重影响页面的加载时间，尤其是当这样的事件在页面加载时发生多次的时候。

为了尽量减小阻塞带来的影响，你可以将script元素放到页面的尾部，在\</body>之前，这样就没有可以被脚本阻塞的元素了。此时，页面中的其它组件（文件）已经被下载完毕并呈现给用户了。

最坏的“反模式”是在文档的头部使用独立的文件：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
    <!-- ANTIPATTERN -->
    <script src="jquery.js"></script>
    <script src="jquery.quickselect.js"></script>
    <script src="jquery.lightbox.js"></script>
    <script src="myapp.js"></script>
</head>
<body>
    ...
</body>
</html>
```

一个更好的选择是将所有的文件合并起来：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
    <script src="all_20100426.js"></script>
</head>
<body>
    ...
</body>
</html>
```

最好的选择是将合并后的脚本放到页面的尾部：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
</head>
<body>
    ...
    <script src="all_20100426.js"></script>
</body>
</html>
```

HTTP分块

HTTP协议支持“分块编码”。它允许将页面分成一块一块发送。所以如果你有一个很复杂的页面，你不需要将那些每个站都多多少少会有的（静态）头部信息也等到所有的服务端工作都完成后再开始发送。

一个简单的策略是在组装页面其余部分的时候将页面\ 的内容作为第一块发送。也就是像这样子：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
</head>
<!-- end of chunk #1 -->
<body>
    ...
    <script src="all_20100426.js"></script> </body>
</html>
<!-- end of chunk #2 -->
```

这种情况下可以做一个简单的发动，将JavaScript移回\，随着第一块一起发送。

这样的话可以让服务器在拿到head区内容后就开始下载脚本文件，而此时页面的其它部分在服务端还尚未就绪：

```
<!doctype html>
<html>
<head>
    <title>My App</title>
    <script src="all_20100426.js"></script> </body>
</head>
<!-- end of chunk #1 -->
<body>
    ...
</html>
<!-- end of chunk #2 -->
```

一个更好的办法是使用第三块内容，让它在页面尾部，只包含脚本。如果有一些每个页面都用到的静态的头部，也可以将这部分随和一块一起发送：

```
<!doctype html> <html>
<head>
    <title>My App</title> </head>
<body>
    <div id="header">
        <img src="logo.png" />
        ...
    </div>
    <!-- end of chunk #1 -->

    ... The full body of the page ...

    <!-- end of chunk #2 -->
    <script src="all_20100426.js"></script>
</body>
</html>
<!-- end of chunk #3 -->
```

这种方法很适合使用渐进增强思想的网站（关键业务不依赖JavaScript）。当HTML的第二块发送完毕的时候，浏览器已经有了一个加载、显示完毕并且可用的页面，就像禁用JavaScript时的情况。当JavaScript随着第三块到达时，它会进一步增强页面，为页面锦上添花。

动态\元素实现非阻塞下载

前面已经说到过，JavaScript会阻塞后面文件的下载，但有一些模式可以防止阻塞：

- 使用XHR加载脚本，然后作为一个字符串使用eval()来执行。这种方法受同源策略的限制，而且引入了eval()这种“反模式”。
- 使用defer和async属性，但有浏览器兼容性问题
- 使用动态\元素

最后一种是一个很好并且实际可行的模式。和介绍JSONP时所做的一样，创建一个新的script元素，设置它的src属性，然后将它放到页面上。

这是一个异步加载JavaScript，不阻塞其它文件下载的示例：

```
var script = document.createElement("script");
script.src = "all_20100426.js";
document.documentElement.firstChild.appendChild(script);
```

这种模式的缺点是，在这之后加载的脚本不能依赖这个脚本。因为这个脚本是异步加载的，所以无法保证它什么时候会被加载进来，如果要依赖的话，很可能会访问到（因还未加载完毕导致的）未定义的对象。

如果要解决这个问题，可以让内联的脚本不立即执行，而是作为一个函数放到一个数组中。当依赖的脚本加载完毕后，再执行数组中的所有函数。所以一共有三个步骤。

首先，创建一个数组用来存储所有的内联代码，定义的位置尽量靠前：

```
var mynamespace = {
    inline_scripts: []
};
```

然后你需要将这些单独的内联脚本包裹进一个函数中，然后将每个函数放到inline_scripts数组中，也就是这样：

```
// was:
// <script>console.log("I am inline");</script>

// becomes:
<script>
    mynamespace.inline_scripts.push(function () {
        console.log("I am inline");
    });
</script>
```

最后一步是使用异步加载的脚本遍历这个数组，然后执行函数：

```
var i, scripts = mynamespace.inline_scripts, max = scripts.length;
for (i = 0; i < max; max += 1) {
    scripts[i]();
}
```

插入\元素

通常脚本是插入到文档的中的，但其实你可以插入任何元素中，包括body（像JSONP示例中那样）。

在前面的例子中，我们使用documentElement来插到\中，因为documentElement就是\，它的第一个子元素是\：

```
document.documentElement.firstChild.appendChild(script);
```

通常也会这样写：

```
document.getElementsByTagName("head")[0].appendChild(script);
```

当你能控制结构的时候，这样做没有问题，但是如果你在写挂件（widget）或者是广告时，你并不知道托管它的是一个什么样的页面。甚至可能页面上连\和\都没有，尽管document.body在绝大多数没有\标签的时候也可以工作：

```
document.body.appendChild(script);
```

可以肯定页面上一定存在的一个标签是你正在运行的脚本所处的位置——script标签。（对内联或者外部文件来说）如果没有script标签，那么代码就不会运行。可以利用这一事实，在页面的第一个script标签上使用insertBefore()：

```
var first_script = document.getElementsByTagName('script')[0];
first_script.parentNode.insertBefore(script, first_script);
```

frist_script是页面中一定存在的一个script标签，script是你创建的新的script元素。

延迟加载

所谓的延迟加载是指在页面的load事件之后再加载外部文件。通常，将一个大的合并后的文件分成两部分是有好处的：

- 一部分是页面初始化和绑定UI元素的事件处理函数必须的
- 第二部分是只在用户交互或者其它条件下才会用到的

目标就是逐步加载页面，让用户尽快可以进行一些操作。剩余的部分可以在用户可以看到页面的时候再在后台加载。

加载第二部分JavaScript的方法也是使用动态script元素，将它加在head或者body中：

```
    .. The full body of the page ...

    <!-- end of chunk #2 -->
    <script src="all_20100426.js"></script>
    <script>
    window.onload = function () {
        var script = document.createElement("script");
        script.src = "all_lazy_20100426.js";
        document.documentElement.firstChild.appendChild(script);
    };
    </script>
</body>
</html>
<!-- end of chunk #3 -->
```

对很多应用来说，延迟加载的部分大部分情况下会比核心部分要大，因为我们关注的“行为”（比如拖放、XHR、动画）只在用户初始化之后才会发生。

按需加载

前面的模式会在页面加载后无条件加载其它的JavaScript，并假设这些代码很可能会被用到。但我们是否可以做得更好，分部分加载，在真正需要使用的时候才加载那一部分？

假设你页面的侧边栏上有一些tabs。点击tab会发出一个XHR请求获取内容，然后更新tab的内容，然后有一个更新的动画。如果这是页面上唯一需要XHR和动画库的地方，而用户又不点击tab的话会怎样？

下面介绍按需加载模式。你可以创建一个require()函数或者方法，它接受一个需要被加载的脚本文件的文件名，还有一个在脚本被加载完毕后执行的回调函数。

require()函数可以被这样使用：

```
require("extra.js", function () {
    functionDefinedInExtraJS();
});
```

我们来看一下如何实现这样一个函数。加载脚本很简单——你只需要按照动态\元素模式做就可以了。获知脚本已经加载需要一点点技巧，因为浏览器之间有差异：

function require(file, callback) {

```
var script = document.getElementsByTagName('script')[0], newjs = document.createElement('

// IE
newjs.onreadystatechange = function () {
    if (newjs.readyState === 'loaded' || newjs.readyState === 'complete') {
        newjs.onreadystatechange = null;
        callback();
    }
};

// others
newjs.onload = function () {
    callback();
};

newjs.src = file;
script.parentNode.insertBefore(newjs, script);
```

}

这个实现的几点说明：

- 在IE中需要订阅readystatechange事件，然后判断状态是否为“loaded”或者“complete”。其它的浏览器会忽略这里。
- 在Firefox，Safari和Opera中，通过onload属性订阅load事件。
- 这个方法在Safari 2中无效。如果必须要处理这个浏览器，需要设一个定时器，周期性地去检查某个指定的变量（在脚本中定义的）是否有定义。当它变成已定义时，就意味着新的脚本已经被加载并执行。

你可以通过建立一个人为延迟的脚本来测试这个实现（模拟网络延迟），比如ondemand.js.php，如：

```
<?php
header('Content-Type: application/javascript');
sleep(1);
?>
function extraFunction(logthis) {
    console.log('loaded and executed');
    console.log(logthis);
}
```

现在测试require()函数：

```
require('ondemand.js.php', function () {
    extraFunction('loaded from the parent page');
    document.body.appendChild(document.createTextNode('done!'));
});
```

这段代码会在console中打印两条，然后页面中会显示“done!”，你可以在http://jspatterns.com/book/7/ondemand.html看到示例。

预加载JavaScript

在延迟加载模式和按需加载模式中，我们加载了当前页面需要用到的脚本。除此之外，我们也可以加载当前页面不需要但可能在接下来的页面中需要的脚本。这样的话，当用户进入第二个页面时，脚本已经被预加载过，整体体验会变得更快。

预加载可以简单地通过动态脚本模式实现。但这也意味着脚本会被解析和执行。解析仅仅会在页面加载时间中增加预加载消耗的时间，但执行却可能导致JavaScript错误，因为预加载的脚本会假设自己运行在第二个页面上，比如找一个特写的DOM节点就可能出错。

仅加载脚本而不解析和执行是可能的，这也同样适用于CSS和图像。

在IE中，你可以使用熟悉的图片信标模式来发起请求：

```
new Image().src = "preloadme.js";
```

在其它的浏览器中，你可以使用\替代script元素，然后将它的data属性指向脚本的URL：

```
var obj = document.createElement('object');
obj.data = "preloadme.js";
document.body.appendChild(obj);
```

为了阻止object可见，你应该设置它的width和height属性为0。

你可以创建一个通用的preload()函数或者方法，使用条件初始化模式（第4章）来处理浏览器差异：

```
var preload;
if (/*@cc_on!@*/false) { // IE sniffing with conditional comments
    preload = function (file) {
        new Image().src = file;
    };
} else {
    preload = function (file) {
        var obj = document.createElement('object'),
            body = document.body;

        obj.width = 0;
        obj.height = 0;
        obj.data = file;
        body.appendChild(obj);
    };
}
```

使用这个新函数：

```
preload('my_web_worker.js');
```

这种模式的坏处在于存在用户代理（浏览器）嗅探，但这里无法避免，因为特性检测没有办法告知足够的浏览器行为信息。比如在这个模式中，理论上你可以测试typeof Image是否是“function”来代替嗅探。但这种方法其实没有作用，因为所有的浏览器都支持new Image();只是有一些浏览器会为图片单独做缓存，意味着作为图片缓存下来的组件（文件）在第二个页面中不会被作为脚本取出来，而是会重新下载。

> 浏览器嗅探中使用条件注释很有意思，这明显比在navigator.userAgent中找字符串要安全得多，因为用户可以很容易地修改这些字符串。 比如： var isIE = /*@cc_on!@*/false; 会在其它的浏览器中将isIE设为false（因为忽略了注释），但在IE中会是true，因为在条件注释中有取反运算符!。在IE中就像是这样： var isIE = !false; // true

预加载模式可以被用于各种组件（文件），而不仅仅是脚本。比如在登录页就很有用。当用户开始输入用户名时，你可以使用打字的时间开始预加载（非敏感的东西），因为用户很可能会到第二个也就是登录后的页面。

小结

在前一章中我们讨论了JavaScript核心的模式，它们与环境无关，这一章主要关注了只在客户端浏览器环境中应用的模式。

我们看了：

- 分离的思想（HTML：内容，CSS：表现，JavaScript：行为），只用于增强体验的JavaScript以及基于特性检测的浏览器探测。（尽管在本章的最后你看到了如何打破这个模式。）
- DOM编程——加速DOM访问和操作的模式，主要通过将DOM操作集中在一起来实现，因为频繁和DOM打交道代码是很高的。

- 事件，跨浏览器的事件处理，以及使用事件代码来减少事件处理函数的绑定数量以提高性能。
- 两种处理长时间大计算量脚本的模式——使用setTimeout()将长时间操作拆分为小块执行和在现代浏览器中使用web workers。
- 多种用于远程编程，进行服务器和客户端通讯的模式——XHR，JSONP，框架和图片信标。
- 在生产环境中部署JavaScript的步骤——将脚本合并为更少的文件，压缩和gzip（总共节省85%），可能的话托管到CDN并发送Expires头来提升缓存效果。
- 基于性能考虑引入页面脚本的模式，包括：放置\元素的位置，同时也可以从HTTP分块获益。为了减少页面初始化时加载大的脚本文件引起的初始化工作量，我们讨论了几种不同的模式，比如延迟加载、预加载和按需加载。